New Applications of SPQR-Trees in Graph Drawing

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

René Weiskircher

Datum des Kolloqiums: **23. Mai 2002**
Dekan der Naturwissenschaftlich-Technischen Fakultät I:
Prof. Dr. Philipp Slusallek
Gutachter:
Prof. Dr. Petra Mutzel, Technische Universität Wien, Österreich
Prof. Dr. Kurt Mehlhorn, Max–Planck–Insititut für Informatik, Saarbrücken

Abstract

We study two problems that arise in the field of graph drawing. In both problems, we have to optimize a function over the set of all embeddings of a planar graph. The algorithms we have developed to solve the problems rely on the data structure SPQR-tree which represents the set of all embeddings of a planar biconnected graph. The first problem we consider is the problem of inserting an additional edge into a planar graph with the minimum number of crossings. We have found a linear time algorithm to solve this previously unsolved problem to optimality. The second problem is finding an orthogonal drawing of a planar biconnected graph with the minimum number of bends. This problem is known to be NP-hard. Using the SPQR-tree, we have developed an integer linear program that is constructed recursively and represents all embeddings of a graph. By combining it with a linear program that describes all orthogonal representations of a graph for a fixed embedding, we have constructed a mixed integer linear program where an optimal solution corresponds to an orthogonal representation with the minimum number of bends over all embeddings of the graph. Solving this program with a commercial mixed integer solver is for large graphs faster than using a previously known branch and bound algorithm for the bend number minimization problem.

Kurzzusammenfassung

Wir untersuchen zwei Probleme auf dem Gebiet des Zeichnens von Graphen. Bei beiden Problemen geht es darum, eine Funktion über der Menge aller Einbettungen eines planaren Graphen zu optimieren. Die Algorithmen, die wir für beide Probleme entwickelt haben, benutzen SPQR-Bäume. Ein solcher Baum repräsentiert die Menge aller Einbettungen eines planaren Graphen. Das erste von uns betrachtete Problem ist das Einfügen einer zusätzlichen Kante in einen planaren Graphen mit möglichst wenigen Kreuzungen. Wir haben einen Algorithmus mit linearer Laufzeit entwickelt, der dieses Problem optimal löst. Das zweite Problem ist das Berechnen einer orthogonalen Zeichnung mit der minimalen Anzahl von Knicken für einen planaren zweizusammenhängenden Graphen. Es ist bekannt, dass dieses Problem NP-schwer ist. Wir benutzen den SPQR-Baum, um ein ganzzahliges lineares Programm zu entwickeln. Dieses wird rekursiv berechnet und seine Lösungen entsprechen den Einbettungen des Graphen. Dieses ganzzahlige lineare Programm haben wir mit einem linearen Programm kombiniert, bei dem eine optimale Lösung einer orthogonalen Repräsentation des Graphen mit der minimalen Knickanzahl für eine feste Einbettung entspricht. Das Resultat ist ein gemischt-ganzzahliges lineares Programm, bei dem eine optimale Lösung einer orthogonalen Repräsentation des Graphen entspricht, die knickminimal über alle möglichen Einbettungen des Graphen ist. Wir Lösen dieses Programmes mittels eines kommerziellen Lösers für gemischt-ganzzahlige Programme. Unsere Experimente zeigen, dass dieses Vorgehen für große Graphen schneller zum Ziel führt als das Lösen des selben Problems mit Hilfe eines bekannten Branch & Bound Algorithmus.

SPQR
FONDERIE E OFFICINE
G. BERTA - FIRENZE

Acknowledgments

First and foremost I want to thank my advisor Petra Mutzel for believing in me and giving me optimal support during the time I have spent working on this thesis. She introduced me into the fields of graph drawing and combinatorial optimization and her advice in all matters connected to this thesis was invaluable.

I also want to thank Kurt Mehlhorn because he gave me the possibility to work at the Max-Planck-Institute for computer science in Saarbrücken during the first and most important two years of my thesis. The Max-Planck-Institute is an ideal place to work in and most of what I know about algorithms I learned in the lectures of Kurt who is a wonderful teacher.

I had many fruitful scientific discussions with Gunnar Klau, who has shared an office with me since I started with my thesis. He also helped a lot in making the four years an enjoyable experience. I could not have constructed the algorithm for inserting an edge into a planar graph without Carsten Gutwenger, who knows more about SPQR-trees than I do. Karsten Klein made it possible that I can produce drawings with my bend minimization program because he programmed the interface to the AGD. I also have to thank Graham Farr for providing the example that shows the difference between crossing minimization and the insertion of an edge into a planar graph with the minimum number of crossings.

I learned a lot about crossing minimization from Thomas Ziegler who wrote his PhD thesis about this topic. Walter Didimo, who is one of the inventors of the first algorithm for minimizing the number of bends over all embeddings, gave me access to GDToolkit and made the comparison of their method with my algorithm possible. I also have to thank Friedrich Eisenbrand for important advice.

I am indebted to the people in the Algorithms and Data Structure group at the Technical University in Vienna. The group seminars helped me to clarify my thoughts and provided valuable ideas. In particular, I want to thank Martin Gruber for his quick response whenever something went wrong with our computer systems.

The “Graduiertenkolleg Effizienz und Komplexität von Algorithmen und Rechenanlagen” financed me during the first year of work on my thesis. I also want to thank the people who took it upon them to proofread this thesis: Christoph Buchheim, Gunnar Klau and Merijam Percan.

Contents

Chapter 1

Introduction

A graph is a mathematical structure often used to model the relationships between objects. It consists of *vertices* and pairs of vertices that are called *edges*. One way of defining a graph is as a list of its vertices and edges. We can also describe it as a two-dimensional matrix indexed by the vertices where the entries are the edges. But in most cases, a drawing of the graph conveys the information stored in it much easier to a human than a textual representation.

The most common way of drawing a graph is the following: The vertices are drawn as shapes in the plane and the edges as lines connecting these shapes. A point where two of the lines meet is called a *crossing* if it is not the representation of a vertex. Graphs that can be drawn in the plane without any crossings are called *planar graphs* and the drawings without crossings *planar drawings*.

The set of different planar drawings of a planar graph is infinite, but it can be divided into a finite set of equivalence classes of drawings called the *embeddings* of the graph. We say a drawing *realizes* a certain embedding Π if it belongs to the equivalence class Π. All drawings that realize the same embedding are in a topological sense equivalent. One way of defining an embedding is by the circular sequence of the edges around each vertex in clockwise order. See Figure 1.1(a) and 1.1(b) for a small example of two different embeddings of a graph.

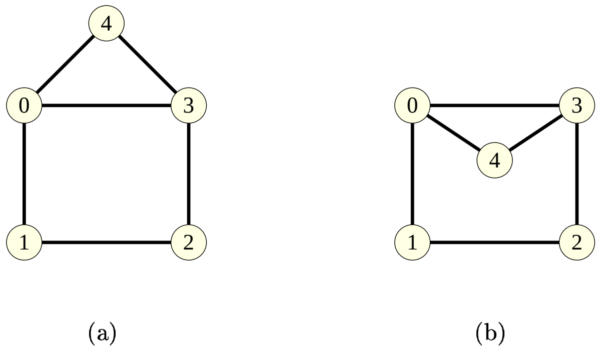


Figure 1.1: Two different drawings of the same planar graph that realize different embeddings.

The data structure *SPQR-tree* represents the decomposition of a planar biconnected graph into its triconnected components. We can easily enumerate the embeddings of each of these components. The combination of the embeddings of the components defines an embedding of the original graph. Therefore, we can use the SPQR-tree to enumerate all embeddings of a biconnected planar graph. By fixing only the embeddings of some of the components in the SPQR-tree, the tree can represent a set of embeddings.

SPQR-trees are so useful because there are many interesting problems concerning planar graphs that are easy to solve if the embedding of the graph is given but more difficult if the embedding is not part of the input. These problems are also easy to solve for triconnected graphs, because they have only two different embeddings that are mirror images of each other. In contrast, a biconnected graph can have an exponential number of different embeddings.

Now assume that we want to solve a problem for biconnected planar graphs that we can easily solve if the embedding is fixed. It follows that we can solve it for the triconnected components of the SPQR-tree of that graph. If we can find a way to combine the solutions for the components to construct a solution for the original graph, we have found an algorithm for solving the problem for biconnected graphs where the embedding is not fixed.

We used this approach to solve the following problem: Given a planar graph G and an additional edge e. Let G' be the graph G together with the new edge e. We want to find a drawing of G' with the minimum number of crossings under the condition that edges of G do not cross each other. This problem is part of the *planarization method* for drawing non planar graph.

If the embedding of G is fixed, this problem can be solved quickly and easily by computing a shortest path in the slightly modified dual graph of G. This graph is defined by the embedding of G. Using the properties of SPQR-trees, we developed an algorithm for solving this problem in the case where the embedding of G is not fixed. We use the algorithm for fixed embeddings on the triconnected components in the SPQR-tree and then combine the solutions to construct a solution for G. Our new method runs in linear time just like the method for fixed embeddings.

Here is another way of using the properties of SPQR-trees to solve a problem for biconnected planar graphs if we can solve it for a fixed embedding: Assume that there is a linear program or integer linear program L_1 that models the problem for the case where the embedding is fixed. We have developed a method for computing an integer linear program L_2 where the set of solutions corresponds to the set of embeddings of a planar biconnected graph. If we combine L_1 and L_2, we have an integer linear or mixed integer linear program that can be used to solve the original problem for biconnected planar graphs if the embedding is not part of the input.

This is the approach that we used for the bend-minimization problem. The input for this problem is a biconnected planar graph G and we want to find an orthogonal drawing of G where the number of bends is minimal. In an orthogonal drawing, all edges are drawn as sequences of horizontal and vertical line segments. A bend is a point where a vertical and a horizontal segment of the same edge meet.

There is a well known algorithm for solving this problem if the embedding of G is fixed. It has been shown that the problem is NP-hard if the embedding is not fixed. The algorithm for a given embedding constructs a minimum cost flow network where the optimum solution corresponds to a drawing of G with the minimum number of bends. This minimum cost flow network can easily be transformed into a linear program.

We combined this linear program with our new integer linear program that describes all embeddings of a graph. The result is a mixed integer linear program where the optimum solution corresponds to a drawing of G with the minimum number of bends over all embeddings. We use a commercial mixed integer solver to compute the solutions. Our computational results show that this approach outperforms the previously known branch & bound method if the graph exceeds a certain size.

1.1 Graph Drawing

Graph drawing is the art of visualizing graphs. The goal is to generate a picture where the user can easily read off which vertices are connected by an edge. This thesis deals with several aspects in the field of automatic graph drawing, where the goal is to let the computer generate a good picture of a graph. A good overview of algorithms used in graph drawing has been written by Battista, Eades, Tamassia, and Tollis (1). Another survey on this topic has been edited by Kaufmann and Wagner (36).

There are many fields of science where graph drawings are used to visualize relationships between objects. The reason is that almost all scientific fields are dealing with complex systems of interacting entities. To understand these systems, it is necessary to understand the relationships between the objects.

In social science, the researchers are often working on networks of people in a group like the workers in a company. These relationships are usually visualized in a drawing where the actors are drawn as shapes of different types, depending on their function. The relationships are drawn as lines of different color and style, depending on the nature of the relationship.

There are also several applications of graph drawing in economic sciences. Business processes are often visualized as graph drawings where the basic tasks are represented by the vertices and there is an edge from one vertex to another if the task represented by the first vertex has to be executed before the task represented by the second vertex.

Another application in the field of economic sciences is the drawing of organization charts where the relationships between organizational units are visualized. A similar application are diagrams that show the ownership relations between companies (see Figure 1.2).

In software engineering, graph drawing plays an important role in the design of information systems and in reverse engineering. Examples are call graphs of functions and class diagrams. The vertices in a call graph represent functions and there is an edge from one vertex to another if the function represented by the first vertex calls the function represented by the second vertex. In class diagrams, the vertices represent the classes in an object oriented design. The edges can represent class inheritance or the fact that a method of one class calls the method of another class.

Another area where graphs are drawn is in database design. In this field, designers use a graphical language called *entity relationship model* (ERM). The objects in the database are

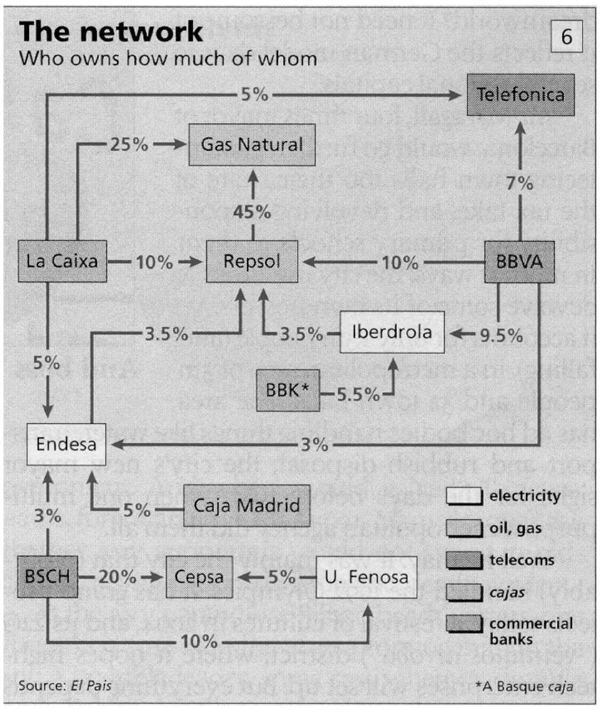


Figure 1.2: A graph that shows ownership relations of Spanish companies (33).

drawn as different shapes and lines connecting these shapes show connections between the objects.

There are many different ways to draw a graph. Depending on the application, a drawing has to meet a set of requirements. A set of requirements is called a *drawing convention*. As an example, we may want that all vertices have the same shape and size and that all edges are drawn as straight line segments. Or we may allow different vertex sizes and all edges have to be polylines consisting of a sequence of vertical and horizontal line segments.

Not every drawing that satisfies a drawing convention for a particular application is a good drawing. To be easily readable, a graph drawing should satisfy aesthetic criteria, too. The following criteria are the most important for almost all applications:

- **Small number of crossings.** If a drawing contains many points where edges that are not connected to the same vertex meet, then it becomes quite difficult to follow the edges. The reader needs more time to figure out which vertices are connected. For most applications, this criterion is considered the most important. See Figure 1.3 for an example that shows two drawings of the same graph, one with many crossings and one with only one crossing.

- **Small number of bends.** If edges are drawn as polylines consisting of horizontal and vertical line segments (like in Figure 1.2), it is desirable that each edge consists only of a small number of segments. If an edge consists of many segments, the reader will find it

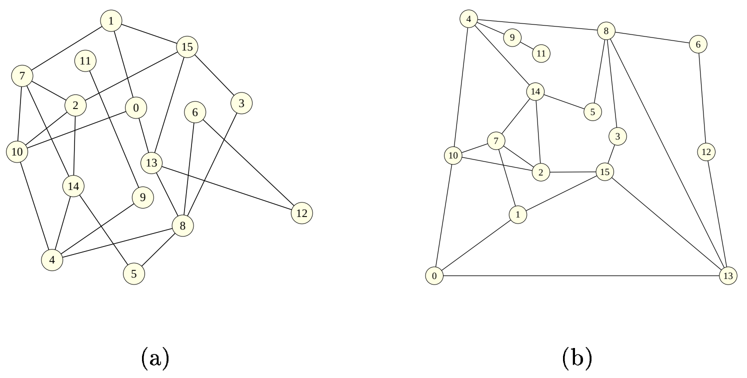


Figure 1.3: Two drawings of the same graph with different number of crossings.

difficult to locate the vertices connected by the edge. Figure 1.4 shows two drawings of the same graph with different number of bends.

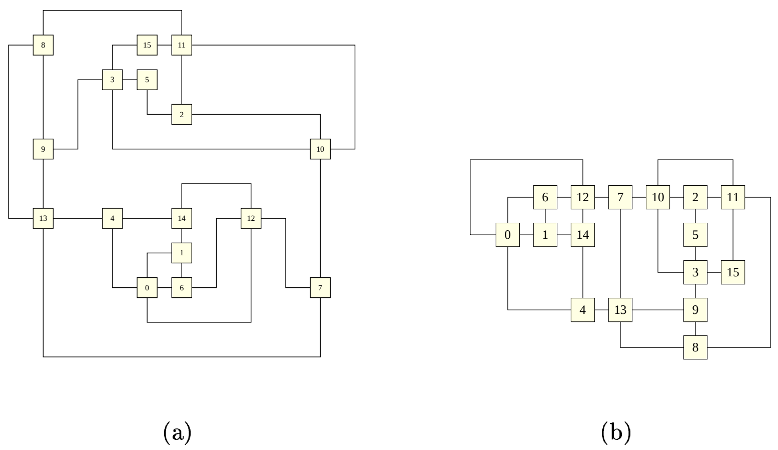


Figure 1.4: Two drawings of the same graph with different numbers of bends.

- **Small drawing area.** If we generate a drawing that is to be displayed on a computer screen or printed on a printer, then it is always drawn on a grid. The reason is that all output devices of a computer have limited resolution. It is desirable that the smallest rectangle that encloses the computed drawing covers a small number of grid points. When we scale the picture to fit on the output device, a drawing with small area will show the vertices and edges of the drawing larger then a drawing that covers a large area. This effect is obvious in Figure 1.4. Although the drawing on the left takes up more area of the page, the vertices in the drawing on the right can be drawn larger. The reason is that the drawing on the right can be enclosed in a smaller rectangle when drawn on the same grid as the drawing on the left. We want to draw the edges and vertices as large as possible. Therefore, it is important that the drawing can be enclosed in a small rectangle on the grid.

- **Small edge length.** We want the edges to be short because this implies that related objects are drawn close together. The average edge length should be small as well as the length of the longest edge.

- **Symmetry.** Some graphs can be drawn in such a way that the drawing is symmetric. A symmetric drawing can lead to a deeper understanding of the graph because the geometric symmetry in the drawing corresponds to an automorphism in the graph. It is also desirable to draw a graph such that the drawing is almost symmetric if only a small change in the graph is needed to make a symmetric drawing possible.

There are many other aesthetic criteria, but the ones mentioned above are considered the most important for most applications. The problem is that all these criteria are difficult to optimize and that some may contradict each other. Often, a symmetric drawing has a greater number of crossings than the drawing with the minimum number of crossings. Figure 1.5 shows a primitive example for this case.

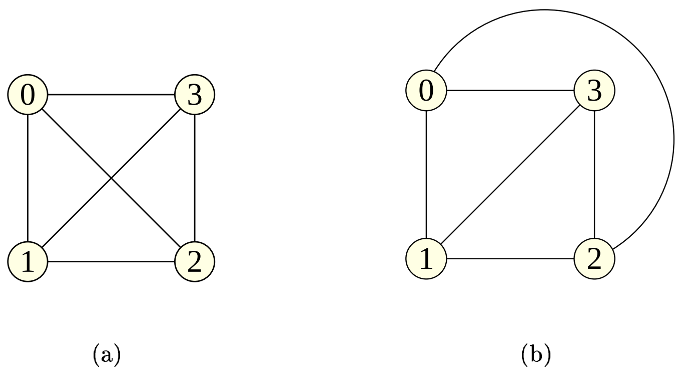


Figure 1.5: Two drawings of the same graph. The drawing on the left maximizes symmetry while the drawing on the right minimizes the number of crossings

In this thesis, we concentrate on the first two aesthetic criteria: the number of crossings and the number of bends in an orthogonal drawing. In Chapter 3, we investigate the problem of inserting an edge into a planar graph such that the number of generated crossings is minimized. The resulting algorithm can be used in a heuristic to draw arbitrary graphs such that the number of crossings is small. In Chapter 4, we develop an algorithm for computing an orthogonal drawing of a biconnected planar graph with the minimum number of bends.

1.2 SPQR-Trees

SPQR-trees were introduced by Di Battista and Tamassia and represent the decomposition of a biconnected graph into triconnected components. They can also be used for enumerating all embeddings of a planar graph. Embeddings of a planar graph describe the topology of a drawing without edge crossings. One way of describing an embedding is by the sequence of incident edges around each vertex in clockwise order. The fact that SPQR-trees can enumerate all embeddings of a biconnected graph motivated us to use them in our attempt to solve the two problems investigated in this thesis.

Let us assume that there is an algorithm for solving a certain problem when the input is a planar biconnected graph and a fixed embedding. Then we can solve the problem for the case where the input is only the planar graph by enumerating all embeddings using the SPQR-tree of the graph and applying the original algorithm to each embedding. The problem with this approach is that the number of different embeddings of a planar graph can be exponential in the size of the graph.

A configuration of the SPQR-tree of a graph G defines an embedding of G. If we only define parts of the configuration of the tree, we can represent partial embeddings of G. A partial embedding represents a subset of all embeddings of G. Assume that we want to optimize some function over the set of all embeddings. If we can compute bounds for the value of this function for the set of embeddings represented by a partial embedding, we can construct a branch and bound algorithm for the optimization problem. This approach is used by Bertolazzi, Battista, and Didimo (9) in the construction of a branch and bound algorithm for the bend minimization problem in orthogonal drawings.

1.3 Application of SPQR-Trees in this Thesis

SPQR-trees decompose a biconnected graph into triconnected components. Assume that we are investigating a problem that can be solved quickly using a certain algorithm for the case where the input graph is triconnected. As we have already stated at the beginning of this introduction, one way of solving such a problem for biconnected graphs is to apply the known algorithm to the triconnected components of the graph and then to merge the obtained solutions into a solution for the original graph using the structure of the SPQR-tree.

This is the approach we used for the problem of inserting an edge into a planar graph G with the minimum number of crossings. If G is triconnected or the embedding of G is fixed, we can use a well known linear time algorithm to solve the problem to optimality. This algorithm finds a shortest path in a modified dual graph of G and translates this path into a list of crossings that are needed to insert the edge.

The algorithm finds the optimum solution in the triconnected case because the embeddings of a triconnected graph are all equivalent with respect to the insertion of a new edge. We do not have to consider all possible embeddings to compute a solution with the minimum number of crossings. This is not the case when the graph is not triconnected.

Before we investigated the problem, there was no polynomial algorithm known to solve the problem for the case where the original graph is not triconnected. Our result is that it suffices to apply the algorithm for fixed embeddings to certain subgraphs of the original graph that are defined by the SPQR-tree. This enables us to solve the problem in linear time.

The situation is similar for the bend minimization problem. If the embedding of a graph is fixed, there is a well known algorithm based on network flows to compute an orthogonal drawing of the graph with the minimum number of bends. Furthermore, it is known that the problem is NP-hard if the embedding is not fixed (24).

As already mentioned, there is a branch and bound algorithm for solving the problem for biconnected graphs in the case where the embedding is not fixed (9). We want to apply integer linear programming methods to the problem.

Our first step was to find an integer linear program where the set of solutions corresponds to the set of embeddings of the graph. We achieved this by first defining this integer linear program for the triconnected components of the graph. Then we used the structure of the SPQR-tree to recursively combine the programs into a program where the set of solutions corresponds to the embeddings of the whole graph. This is the first integer linear program formulation of the set of embeddings of a graph.

In the second step, we combined this integer linear program with a linear program where the optimum solution describes an orthogonal representation of the graph with the minimum number of bends for a fixed embedding. The result is a mixed integer linear program where an optimum solution corresponds to an orthogonal representation of the graph with the minimum number of bends over all embeddings of the graph. We solve this program using a commercial mixed integer solver.

1.4 Inserting an Edge into a Planar Graph

One of the most important aesthetic criteria in graph drawing is the number of edge crossings. A drawing with a large number of crossings is not easily readable. Unfortunately, it is in general NP-complete to decide if a graph can be drawn with k crossings for a positive integer k (23). There is no algorithm known for computing the crossing number (the minimum number of crossings in a drawing of the graph) even for graphs of moderate size (say 20 vertices) in reasonable time.

Therefore, heuristics are used to compute drawings of graphs with few crossings. One of the best methods is known as the *planarization method.* It works in two steps: First, a preferably small number of edges is deleted from the graph to make it planar (drawable without crossings). Then the deleted edges are inserted one by one, hopefully generating only a small number of crossings.

The insertion step always inserts one edge into a planar graph such that the edges of the planar graph do not cross in the resulting drawing. If crossings are generated, they all involve the inserted edge. After inserting the edge, the crossings are replaced by dummy vertices of degree four to produce a planar graph. An example for this step is shown in Figure 1.6. Then the next edge can be inserted in the same way. In a post-processing step, the dummy vertices are replaced by real crossings.

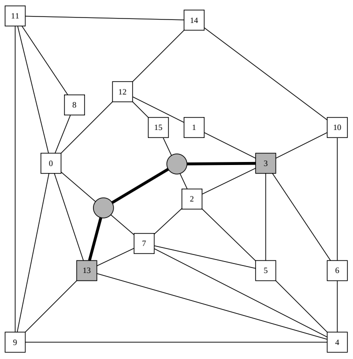


Figure 1.6: Dummy vertices (drawn as grey disks) have been inserted into the graph to get rid of crossings generated by inserting the edge connecting the vertices drawn as grey squares.

Before we investigated the problem, the edge insertion step was performed by choosing an arbitrary embedding of the planar graph and then using a shortest path algorithm on the dual graph. The resulting solution guarantees that the number of crossings generated is minimal for the chosen embedding but not for all possible embeddings.

Our new approach consists of two algorithms. The first algorithm is used if the problem graph is biconnected. The second algorithm is used if the graph is connected but not biconnected. The algorithm for connected graphs calls the algorithm for biconnected graphs as a subroutine. If the graph is not connected and the two vertices of the new edge are in different connected components, it is easy to find an embedding where the new edge can be inserted without producing crossings. Otherwise, we can use the algorithm for connected graphs.

Our algorithms do not in fact compute an embedding of the input graph, but they compute a list of edges that have to be crossed by the new edge. We call this list of edges an *edge insertion path*. To compute the embedding, a linear time post-processing step is needed: We insert dummy vertices on all the edges in the computed list and connect them in the order given by the list. The dummy vertex on the first edge of the edge insertion path is then connected to the first vertex of the new edge and the dummy vertex on the last edge of the list to the second. The result is a path that connects the first vertex of the new edge via the dummy vertices (see Figure 1.6 for an example). Then we can call a linear time embedding algorithm for computing a planar embedding of the resulting graph. Removing the added edges and dummy vertices from this embedding results in an embedding of the original graph.

The algorithm for connected graphs first builds the *block tree* of the graph where the nodes are the vertices and biconnected components of the graph. We use this graph to identify the biconnected components whose embedding may have an influence on the number of crossings when we insert the new edge. We also identify a pair of vertices in each of these components that act as representatives for the two vertices that we want to connect. Then we call the algorithm for biconnected graphs for each of the components as if we wanted to connect the representatives. In the last step, we just have to concatenate the resulting lists of edges to produce an edge insertion path for inserting the new edge.

The algorithm for biconnected graphs works similar to the one for connected graphs. We first compute the SPQR-tree of the graph. Every node in the tree is associated with a special graph called the *skeleton* of the node. The embedding of a skeleton controls part of the embedding of the original graph. By inspecting the SPQR-tree, we can identify the nodes in the tree where the choice of the embedding for the skeleton may have an impact on the number of crossings generated when inserting the new edge. These nodes are on the *critical path* in the tree with respect to the two vertices we want to connect.

It turns out that only one of the four types of nodes in an SPQR-tree are important for the number of crossings, so we can restrict our attention to the nodes of this type on the critical path. For each of the skeletons of these nodes we compute a list of edges that have to be crossed by the new edge. The process works as follows: First, we replace certain edges of the skeleton by subgraphs of the original graph. Then we compute an arbitrary embedding of the resulting graph. Since we have a fixed embedding, we can now compute the dual of this graph. By computing a shortest path in the dual graph, we get the list of edges we want to compute. When we have finished this operation for all the nodes on the critical path, we can just concatenate the computed lists to produce an edge insertion path for inserting the new edge with the minimum number of crossings into the graph.

An important element of the proof of correctness for this approach is that the necessary crossings when inserting an edge into a planar graph only depend on the parts of the graph that are subdivisions of triconnected graphs. A subdivision of a graph is generated by splitting edges and inserting new vertices. If a graph does not contain a subdivision of a triconnected graph, we can insert any edge without crossings.

Another fact about biconnected planar graphs that we use in our algorithm is the following: If a part of the graph, that is connected to the rest of the graph only via a pair of vertices, does not contain any of the two vertices that should be connected, then the embedding of this part has no influence on the number of necessary crossings. This can be seen as follows: If there is an optimal way of inserting the edge such that it does not cross any of the edges in the subgraph, it is obvious that the embedding of this subgraph does not matter. Otherwise, the new edge has to cross a certain number of edges in the subgraph to traverse it. We show via induction that the number of crossings necessary to traverse the subgraph is independent of its embedding.

Using the two facts mentioned above, we can show that the edge insertion path computed by our algorithm has the minimum length among all edge insertion paths. It follows that there is no better way of inserting the edge than to cross all the edges in the computed edge insertion path. This is true for all possible embeddings of the graph.

The running time of the algorithm is linear in the size of the input graph. We do not have to distinguish between the number of vertices or edges in this statement because the number of edges in a planar graph with more than two vertices is is at most $3n-6$ where n is the number of vertices. Note that solving the problem for a fixed embedding by finding the shortest path in the dual graph needs linear time, too. The first reason for the linear running time is that the graphs for which we compute shortest paths in the dual graph during the algorithm are disjoint. The second reason is that the union of all these graphs is only by a constant factor larger than the problem graph.

We wanted to know how the new algorithm performs when used in the planarization framework. Note that although our algorithm inserts one edge optimally into a planar graph, it can not guarantee that repeated use to insert several edges will result in a drawing with the minimum number of crossings. Therefore, it is not self evident that replacing the standard shortest path algorithm in the planarization method with our new algorithm will reduce the number of crossings in the computed drawing if more than one edge has to be inserted.

Fortunately, our computational results on a set of benchmark graphs show that the new algorithm performs better when used in the planarization method then the standard algorithm that chooses an arbitrary embedding and inserts the edge by computing a shortest path in the dual graph. On average, using our new algorithm results in 14 percent less crossings than using the standard algorithm. There are a few graphs where the new approach results in more crossings, but these cases are rare and the difference in the number of crossings is small. Since the asymptotic running time for the standard planarization method is the same as the asymptotic running time for our new approach, the new algorithm can be applied wherever the standard method is used.

Our new algorithm does not solve the crossing minimization problem even if only one edge has to be deleted from the graph to make it planar. Graham Farr (20) has found a graph where inserting one particular edge destroys planarity and where our algorithm computes a drawing where the crossing number is an arbitrary high multiple of the crossing number of the resulting graph.

The reason for this fact is that our algorithm computes a drawing of the graph together with the additional edge with the minimum number of crossings under the constraint that all other edges in the graph do not cross each other. This means that removing the additional edge from the computed drawing results in a planar drawing. If we remove this constraint, it is sometimes possible to construct a drawing with fewer crossings by letting other edges in the graph cross each other.

This is the case in the set of graphs found by Graham Farr. If the graph without the additional edge must be drawn without crossings, we need a number of crossings that grows with the size of the graph. If this constraint is removed, we can always construct a drawing with just two crossings, independent of the size of the graph.

1.5 Bend Minimization in Orthogonal Drawings

In an orthogonal drawing of a graph, all edges are drawn as sequences of horizontal and vertical line segments. An example is the drawing in Figure 1.2 on page 4 that shows the relationships between Spanish companies. This is the preferred method for drawing graphs like entity relationship models that are used in database design. The point where a horizontal line segment meets a vertical line segment of the same edge is called a *bend*. In an orthogonal drawing with a high number of bends, it is difficult to follow the edges and to find out if two vertices are connected by an edge. Therefore, it is important to draw a graph with a small number of bends (see Figure 1.4 on page 5 for an example of a drawing with many bends and a drawing of the same graph with a small number of bends).

It is well known that an orthogonal drawing of a graph with the minimum number of bends is computable in polynomial time if the embedding is fixed. This can be done by transforming the problem into a minimum cost flow problem. The first such algorithm was described by Tamassia (47). If the embedding of the graph is not fixed, it is NP-complete to decide if the graph can be drawn with a certain number of bends (24).

There is a branch and bound algorithm for finding an orthogonal drawing of a planar biconnected graph with the minimum number of bends (9). In this thesis, we attack the same problem using integer linear programming methods. To achieve this, we first developed an integer linear program (ILP) where the solutions correspond to the set of all embeddings of a planar biconnected graph. Then we combined this ILP with a linear program that describes all orthogonal representations of a graph for a fixed embedding. The resulting mixed integer linear program describes all orthogonal representations of a biconnected planar graph and can be used to compute an orthogonal representation with the minimum number of bends.

Again, we use the property of SPQR-trees that they can represent all embeddings of a planar biconnected graph. It is easy to construct ILPs that describe all embeddings of the skeletons in an SPQR-tree. The reason is that skeletons are either triconnected or have a simple structure. Therefore, we can easily describe the set of all embeddings of a skeleton as the solutions of an ILP. The binary variables in these programs correspond to directed cycles in the skeletons.

We have found a way of combining the ILPs computed for components of a graph. The resulting ILP describes the set of all embeddings of the original graph. This enables us to compute an ILP for the original graph recursively. We get the program for the original graph by combining the ILPs for certain components of the graph recursively. The recursion stops at the skeletons of the nodes in the SPQR-tree.

We combine the ILPs of the components by lifting their constraints. Lifting is the process of computing the coefficients for the variables of a higher dimensional problem in a constraint computed for a lower dimensional problem. This enables us to reuse constraints computed for lower dimensional problems in a higher dimensional problem. In our case, we also need some additional constraints when we combine the ILPs of subproblems to compute the ILP for the original problem.

Another way of looking at this process is the following: First, we compute ILPs for the skeletons of the nodes in the SPQR-tree of the graph. Then we combine these ILPs to form the ILPs of larger units of the graph and repeat the process until we have constructed an ILP for the whole graph. In each combination step, we have to lift all previously computed constraints and add new constraints, that form the connection between the smaller ILPs. To our knowledge, this is the first time that such a recursive approach was used to compute an ILP.

An additional problem that we had to solve was that the set of embeddings for one particular type of skeletons in the SPQR-tree can be transformed into the set of all Hamilton tours in a corresponding complete directed graph. Thus, we established a connection to the Asymmetric Traveling Salesman Problem (ATSP). We used the same linear programming formulation often used for the ATSP. Therefore, we have to deal with the problem that the number of constraints may be exponential in the size of the graph.

To cope with this problem, we developed a mechanism that enables us to test efficiently for any integer vector with the right dimension if it violates any of the constraints derived from the ATSP-problem. So we do not have to add these constraints to the ILP. Instead, we test for each solution that we compute, if it violates one of the constraints. If this is the case, we add the constraint and compute a new solution. Our computational results show that this is seldom necessary.

To solve our original problem of minimizing the number of bends in an orthogonal representation, we combined the ILP that describes all embeddings of a graph with a linear program that minimizes the number of bends in an orthogonal representation for a fixed embedding. The linear program is derived from a formulation of the bend minimization problem for fixed embeddings as a minimum cost flow problem. The flow network was already used by Bertolazzi, Battista, and Didimo (9).

Combining the linear program for bend minimization with the ILP describing all embeddings results in a mixed integer program where an optimal solution corresponds to an orthogonal representation of the input graph with the minimum number of bends. We solve this program using CPLEX (34), a commercial system for solving mixed integer problems.

Our computational results show that our new approach to bend minimization is superior to the branch and bound approach described by Bertolazzi, Battista, and Didimo (9) for large graphs with many embeddings. They also show that using an exact algorithm for minimizing the bends in a drawing over all embeddings saves a significant number of bends compared with picking an arbitrary embedding for almost half the graphs we tested.

1.6 Overview

Chapter 2 introduces the data structure SPQR-tree. In Section 2.1, we give a short overview of the applications and features of the trees. Then we introduce some basic concepts that are necessary for the formal definition (Section 2.2). Most of them are concerned with the connectivity of a graph. The definition of SPQR-trees we give in Section 2.3 is slightly different from the definitions used by other authors because we felt that we could get rid of an anomaly in the definition of the root of the tree by first defining a data structure we called Proto-SPQR-tree. The SPQR-tree is then defined as an extension of the Proto-SPQR-tree.

After giving an example for the construction of an SPQR-tree, we present a few properties of the trees that are important in the rest of the thesis in Section 2.4. The topic of Section 2.5 is the connection between the embeddings of a graph and the embeddings of the skeletons of its SPQR-tree. This is the basis for the algorithms presented in this thesis.

Chapter 3 deals with the problem of inserting an edge into a planar graph with the minimum number of crossings. In Section 3.1, we give an introduction to the problem. We present the planarization method where the problem arises and give some results on related crossing minimization problems.

Section 3.2 introduces some necessary concepts. We define *edge insertion paths,* a list of edges that defines how to insert an edge into a planar graph by giving the sequence of existing edges that the new edge crosses. Then we present the concept of *traversing cost*. These cost are defined for subgraphs of the original graph that are connected to the rest of the graph via two vertices. The traversing cost can be interpreted as the number of edges that have to be crossed to get from one side of the subgraph to the other side.

In Section 3.3 we present the algorithm that inserts an edge into a planar graph with the minimum number of crossings if the graph is biconnected and prove its correctness. The running time of the algorithm is linear in the size of the original planar graph.

The topic of Section 3.4 is how to solve the problem for connected graphs. We introduce block trees and present the algorithm. It determines the nodes in the block tree that are important for the number of crossings and calls the algorithm for biconnected graphs for the corresponding blocks. Again, we proof correctness and show that the running time of the algorithm is linear. This section also contains the trivial generalization to non-connected graphs.

Computational results are given in Section 3.5. We compare the performance of the standard planarization method with the modified planarization method where the edges are reinserted using our new algorithm. The results show that the new algorithm significantly improves the performance of the planarization method.

In Section 3.6 we examine the difference between general crossing minimization and the insertion of an edge into a planar graph with the minimum number of crossings. We present an example graph that shows that optimal solutions to both problems can differ by an arbitrary number of crossings for the same graph.

The topic of Chapter 4 is the minimization of the number of bends in an orthogonal drawing of a graph. Section 4.1 gives an introduction to the problem with an overview of the chapter. The integer linear program (ILP) describing the embeddings of a planar biconnected graph is the topic of Section 4.2. First, we give an intuition of how the ILP is constructed. Then we define its variables.

Next, we define the ILPs for the case where the SPQR-tree of the graph has only one inner node. We distinguish three cases according to the type of the only inner node. We have to deal with the special problems that occur if the only inner node is a so called *P-node.* In this case, the set of embeddings can be formulated as the set of Hamilton-tours in a complete directed graph and we use the same formulation as the ILP often used for solving instances of the asymmetric traveling salesman problem.

After the definition of the ILPs for SPQR-trees with one inner node, we present the process of computing the ILP for more complex graphs by first splitting the graph into smaller parts and computing the ILPs for these parts. Then the ILPs are combined to construct the ILP for the original graph. The first part of this process consists of our splitting operation that we use to construct smaller SPQR-trees from a large SPQR-tree. This operation implicitly constructs a number of small graphs from one large graph and is the basis of the recursive construction of the ILP. In the second part of the process, the ILP for the original graph is constructed from the ILPs of the smaller graphs created by the splitting operation.

The proof of correctness of our recursively defined ILP uses structural induction on the SPQR-tree to show that all solutions of the ILP represent embeddings and all embeddings of the graph are solutions.

Section 4.3 deals with the linear program where the solutions correspond to the orthogonal representations of a graph if the embedding is fixed. First, we give an intuition of the contents of the section. Then we define the data structure *orthogonal representation* as an embedding together with two functions that determine the angles between edges adjacent to the same vertex and the bends on each edge. We also give an example for an orthogonal representation.

Orthogonal drawings realize orthogonal representations by assigning lengths to the edge segments. We presents several styles of orthogonal drawings. They differ in the way they treat vertices with degree greater than four. We introduce the *podevsnef* drawing style that we use to draw our graphs. After the presentation of the drawing styles, we show how an orthogonal representation can be formulated as a flow network. We describe how a flow in a special network constructed for a graph and a fixed embedding corresponds to an orthogonal representation of the graph. Then we present a linear program that is equivalent to the flow network.

In Section 4.4, we combine the linear program that describes the orthogonal representations of a graph for a fixed embedding with the ILP that describes all embeddings of a graph. The result is a mixed integer linear program that describes the orthogonal representations of a graph over all its embeddings.

To achieve this, we first have to extend the ILP that describes the embeddings of a graph in such a way that each solution also defines which region of the embedding should be the outer (unbounded) region in a corresponding planar drawing. Then we combine this ILP with the linear program that describes orthogonal representations. This results in the mixed integer linear program where the solutions correspond to all possible orthogonal representations of the graph over all embeddings.

In Section 4.5, we present the algorithm we use to compute an orthogonal representation of a graph with the minimum number of bends. After giving a high level overview of the algorithm, we describe some modifications that speed up the computation. We also develop a preprocessing step that transforms the original graph into a smaller graph. The solution computed for the smaller graph is then transformed into an optimal solution valid for the original graph. Section 4.6 contains the computational results obtained by applying the algorithm to a set of benchmark graphs and comparing its performance with the performance of the algorithm presented by Bertolazzi, Battista, and Didimo (9) and a heuristic. We also present two example graphs together with the resulting drawings.

In Chapter 5, we provide a short overview of the results presented in this thesis and present a few ideas for future research.

Chapter 2

SPQR-Trees

2.1 Introduction

An SPQR-tree is a data structure that represents the decomposition of a biconnected graph with respect to its triconnected components. The data structure was introduced by Battista and Tamassia (3) and is a modification of the structure used by Bienstock and Monma (10). SPQR-trees have been used in many graph algorithms with the main focus on planarity and triconnectivity.

A striking feature of SPQR-trees is that their size is linear in the size of the graph they represent and that they can be computed in linear time. The latter fact relies on the algorithm by Hopcroft and Tarjan (32) for decomposing a graph into triconnected components. There are some errors in the description of this algorithm and only recently Gutwenger and Mutzel (27) succeeded in correcting them and in implementing a correct linear time algorithm for computing SPQR-trees.

SPQR-trees can be made dynamic in the sense that they can be updated efficiently when vertices or edges are added to the graph. Therefore, they have been used to solve on-line problems that are connected with planarity and triconnectivity (5; 6). In (4), SPQR-trees are used to maintain the minimum spanning tree and the transitive closure of a graph during edge insertion and vertex insertion operations.

Some of the most interesting applications of SPQR-trees are in the area of graph drawing. Algorithms in this area use the fact that an SPQR-tree implicitly represents all embeddings of a graph and can be used to enumerate them. This property of SPQR-trees is used in a branch and bound algorithm for computing an orthogonal drawing with the minimum number of bends (9). We attack the same problem using SPQR-trees and integer linear programs in Chapter 4.

Another interesting application of SPQR-trees in graph drawing is the symmetric drawing of graphs in three dimensions. In (30), a polynomial algorithm for drawing planar graphs symmetrically in three dimensions is presented. This algorithms uses SPQR-trees to determine a drawing that displays the maximum number of symmetries.

SPQR-trees have also been used in algorithms for clustered graphs. In a cluster planar drawing of a clustered graph, the vertices of each cluster must be contained in a connected region of the plane that does not contain the drawings of edges or vertices that do no belong to the cluster. If such a drawing exists, the graph is called cluster planar. In (18), a linear time algorithm is presented that tests planar connected clustered graphs for cluster planarity using SPQR-trees.

Another area of graph drawing where branch and bound techniques on SPQR-trees have been used is upward planarity. Bertolazzi, Battista, and Didimo (8) presented an algorithm for computing a quasi upward drawing with the minimum number of bends.

2.2 Preliminaries

SPQR-trees are a rather complex data structure and so we first define some basic concepts that are needed in the definition of the structure. Most of these concepts are concerned with the connectivity of graphs.

A *graph* G is a pair $G = (V, E)$ where V is the set of *vertices* and $E \subseteq V \times V$ is the set of edges. A *subgraph* of a graph G is another graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. We are usually dealing with *undirected graphs*, where the edges are unordered pairs of vertices.

A *drawing of a graph* Γ is a function that maps each vertex of G to a point in the plane and each edge (u, v) to a simple open Jordan curve $\Gamma(u, v)$ with endpoints $\Gamma(u)$ and $\Gamma(v)$. A drawing is *planar* if there exists no pair of edges e_1, e_2 with $e_1 \neq e_2$ such that $\Gamma(e_1) \cap \Gamma(e_2) \neq \emptyset$. A graph G is called planar if there exists a planar drawing of G.

We say the two vertices v_1 and v_2 are *connected* by the edge $e = (v_1, v_2)$. Both vertices are *incident* to e, and e is incident to both vertices. The number of edges incident to a vertex is called the *degree* of that vertex. If v_1 and v_2 are connected by an edge, we say they are *adjacent* to each other.

A *multi graph* is a graph that may contain an arbitrary number of edges defined by the same pair of vertices. Thus, the set E is a multi set because it contains several instances of the same pair of vertices. Let v be a vertex in a graph. Then the edge (v, v) is called a *self loop*. We usually do not allow them in the graphs we are dealing with. Unless stated otherwise, all the graphs we use are undirected multi graphs without self loops.

A *path* P of length k in G is a sequence of the form $P = (v_1, e_1, v_2, \ldots, e_k, v_{k+1})$ where $e_i = (v_i, v_{i+1}) \in E$ for $i \in \{1, \ldots, k\}$ and all the vertices v_1 to v_{k+1} are pairwise different. We say that v_1 and v_{k+1} are connected by P. Sometimes, we also denote a path by giving just the edges on the path ($P = (e_1, \ldots, e_k)$) or just the vertices ($P = (v_1, \ldots, v_{k+1})$). Note that the latter form can be ambiguous for multi graphs. A path P connecting the two vertices v and w together with the edge (w, v) not contained in P is called a *cycle*.

We say that a graph is *connected* if all pairs of pairwise different vertices of the graph are connected by a path. A connected graph that does not contain a cycle is called a *tree*. The maximal connected subgraphs of a graph are called the *connected components* of a graph.

We will also need some stronger notions of connectivity of graphs. We say a graph is *k-connected* for $k > 0$, if for any pair of vertices (u, v) there exist k different paths connecting

u and v that are pairwise vertex disjoint. So any two of the k paths share only the vertices u and v. If a graph is 2-connected, we call it *biconnected* and if a graph is 3-connected, we call it *triconnected*. We call the maximal biconnected subgraphs of a graph the *blocks* of the graph.

If G is a graph, then any vertex v whose removal increases the number of connected components of G is called a *cut vertex* of G. Any unordered pair $\{u, v\}$ of vertices with $u \neq v$ whose removal increases the number of connected components of a graph G is called a *separation pair* of G. See Figure 2.1 for an example of cut vertices and separation pairs. From the definition of k-connectivity it follows easily that a biconnected graph has no cut vertex and a triconnected graph has no separation pair.

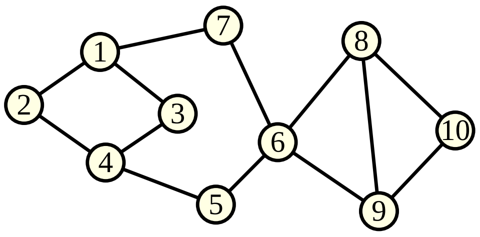


Figure 2.1: Cut vertices and separation pairs: Vertex 6 is a cut vertex while the pairs $\{5, 7\}$ and $\{8, 9\}$ are examples for separation pairs in the graph.

Theorem 2.1 *If a graph G has no multiple edges than it is k-connected for $k \geq 1$ if and only if it contains no set of $k-1$ vertices whose removal splits the graph into at least two connected components.*

Proof The proof follows from Menger's theorem (two different proofs for Menger's theorem can be found in (19)). This theorem says that the minimum number of vertices separating two vertex sets A and B in G is equal to the maximum number of disjoint $A - B$ paths. A vertex set C separates two vertex sets A and B if after the removal of all vertices of C, there is no path left in the graph connecting a vertex in A with a vertex in B.

We assume that A consists of vertex v_1 and B of vertex v_2. First we show that if a graph is k-connected, it does not contain a set of $k-1$ vertices whose removal splits the graph. If G is k-connected, there are at least k vertex disjoint paths connecting v_1 and v_2. By Menger's theorem, the minimum number of vertices separating v_1 and v_2 is k. Since this holds for any pair of vertices v_1 and v_2, there is no set of $k-1$ vertices in G that separate any two vertices in the graph from each other. This is the same as saying that there is no set of $k-1$ vertices whose removal splits the graph.

Now assume that there exists no set of $k-1$ vertices that splits G. So for any two vertices v_1 and v_2 in G, at least k vertices are needed to separate them from each other. By Menger's theorem we know that there are at least k vertex disjoint paths connecting v_1 and v_2. Since this holds for any pair of vertices v_1 and v_2 in G, we know that G is indeed k-connected. $\square$

In the definition of SPQR-trees, we will also need the notion of a *split pair*. A pair $\{u, v\}$ of vertices in a biconnected graph G is a split pair, if $\{u, v\}$ is a separation pair or (u, v) is an edge in G. A *split component* of the split pair $\{u, v\}$ is either an edge that connects u and v together with the vertices u and v or a maximal connected subgraph G' of G such that removing the vertices u and v does not disconnect G'. See Figure 2.2 for an example of the split components of a split pair.

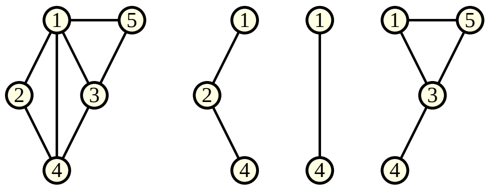


Figure 2.2: A biconnected graph (left) and the three split components of the split pair $\{1, 4\}$ (right)

Observation 2.1 *Let $s = \{v_1, v_2\}$ be a split pair of a biconnected graph G and $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ be two different split components of s. Then the following statements are true:*

$$V_1 \cap V_2 = s \tag{2.1}$$

$$E_1 \cap E_2 = \emptyset \tag{2.2}$$

Proof We start with Equation (2.1). First we show that each split component of s contains s. Assume that S is a split component of s that does not contain both vertices in s. We assume w.l.o.g. that it does not contain v_1. Let v be any vertex in S and $p = (v, u_1, \dots, u_k, v_1)$ a path in G connecting v and v_1. Since G is biconnected, we can always choose p such that none of the inner vertices u_i of p is v_2. We construct the graph S' by adding all the edges and vertices on p to S that are not contained in S.

S is a proper subgraph of S' since S' contains v_1. Removing v_1 and v_2 from S' does not disconnect S' since the path p that we added does not contain v_1 or v_2 as an inner vertex. This is a contradiction to our assumption that S is a split component of s because S is not maximal.

Now we show that no vertex except v_1 and v_2 is contained in more than one split component of s. Assume the two different split components S_1 and S_2 both contain vertex $v \notin s$. Consider the subgraph S that we get by computing the union of the two components S_1 and S_2.

S is connected since both S_1 and S_2 are connected and contain vertex v. Deleting v_1 and v_2 does not disconnect S because it does not disconnect any of the subgraphs S_1 and S_2 of S and both are connected via vertex v with $v \notin \{v_1, v_2\}$. This is a contradiction to the maximality of S_1 and S_2 and completes the proof of Equation (2.1).

Now we prove Equation (2.2). Let e be any edge in G. If it connects v_1 and v_2, the edge gets its own split component by definition. Otherwise, at least one of the vertices of e is not contained in s. We just showed that each vertex except the vertices in s is contained in at most one split component of s and it follows that e is contained in at most one split component of s. This proves Equation (2.2). □

Using Observation 2.2, it is not hard to see that the split components of a biconnected graph form a partition of the edge set. Each edge together with its incident vertices forms a subgraph of G that cannot be disconnected by removing the vertices v_1 and v_2. Therefore, each edge is contained in a maximal subgraph that has this property. So each edge is contained in a split component and because of Observation 2.2 in exactly one split component.

In the definition of SPQR-trees, we will need the notion of the *split graph* of a split pair with respect to some edge (s,t) in a graph G. The split graph of a split pair $\{u,v\}$ is the union of all the split components of G that do not contain the edge (s,t). Informally, the split graph of the split pair $\{u,v\}$ is the part of the original graph that can only be reached from (s,t) via u or v.

In Figure 2.2, the split graph of $\{1,4\}$ with respect to $(3,5)$ is the cycle consisting of the three vertices $1,2$ and 4 together with the incident edges. This is the union of the two split components of $\{1,4\}$ that do not contain edge $(3,5)$.

We say that a split pair $\{u,v\}$ is *dominated* by another split pair $\{x,y\}$ with respect to an edge (s,t) if the split graph of $\{u,v\}$ with respect to (s,t) is a proper subgraph of the split graph of $\{x,y\}$ with respect to (s,t). So if a split pair dominates another split pair with respect to an edge, the dominated split pair can only be reached from the edge by passing a vertex of the dominating split pair.

In Figure 2.2, the pair $\{1,3\}$ is dominated by $\{1,4\}$ with respect to $(2,4)$ because the split graph of $\{1,3\}$ with respect to edge $(2,4)$ is defined by the vertices $1,3$ and 5 and their incident edges while the split graph of $\{1,4\}$ with respect to $(2,4)$ consists of the vertices $1,3,4$ and 5 with their incident edges.

Observation 2.2 *The dominance relation on split pairs with respect to an edge (s,t) in G is not symmetric and not reflexive.*

This observation follows from the fact that two graphs (in this case the split graphs) cannot properly contain each other.

Observation 2.3 *The dominance relation between split pairs in a graph G with respect to an edge (s,t) is transitive.*

This observation follows easily from the transitivity of the subgraph relation.

Using the dominance relation on split pairs, we can define a *maximal split pair* with respect to a certain edge (s,t) in a graph as a split pair that is not dominated by any other split pair in the graph with respect to (s,t). There may be several maximal split pairs in a graph with respect to a certain edge. Consider for example Figure 2.3. The split pairs $\{1,2\}$ and $\{6,9\}$ are both maximal with respect to $(4,5)$. Even split pairs whose split graph is a single edge can be maximal. The split pair $\{5,9\}$ for example is a maximal split pair with respect to $(4,5)$.

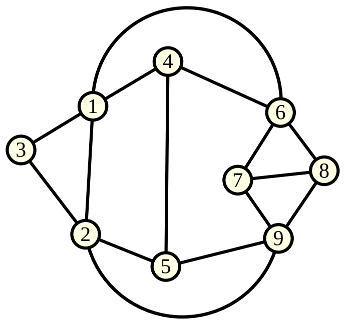


Figure 2.3: The split pairs $\{1, 2\}$ and $\{6, 9\}$ are maximal with respect to edge $(4, 5)$.

2.3 Definition of SPQR-Trees

SPQR-trees are defined for any biconnected graph. The definition is recursive: We first define the root r of the tree and a decomposition of the original graph into subgraphs. The roots of the trees computed for these subgraphs are the children of r.

The definition in this thesis differs slightly from the definitions given in (27; 5; 6; 4) or (9) because we first define a variant of the SPQR-tree called the Proto-SPQR-tree recursively. Then we define the SPQR-tree as a modification of the Proto-SPQR-tree. The reason for this modification is that we are not aware of a clean way to define the root of the SPQR-tree recursively.

The vertices of the SPQR-tree are called *nodes*. The nodes have four different types (S,P,Q, and R) that are explained in the definition below. Each node is associated with a special graph which is called the *skeleton* of that node. We can think of each skeleton as a simplified version of the original graph where some subgraphs have been replaced by single edges.

The tree is constructed by recursively decomposing the original graph into triconnected components. This decomposition starts with an arbitrary edge of the graph which is called the *reference edge* of the decomposition. The node in the SPQR-tree corresponding to this edge will be the root of the tree.

Definition 2.1 (Proto-SPQR-tree) *Let $G = (V, E)$ be a biconnected multi graph (there can be more than one edge connecting a pair of vertices) and $e = (s, t) \in E$ one of its edges. Let G' be the graph G after deletion of edge e ($G' = (V, E - \{e\})$). Then the Proto-SPQR-tree $\mathcal{T}$ with reference edge e is defined as follows:*

- ***Trivial case:*** *G' is a single edge. In this case, $\mathcal{T}$ consists just of one Q-node. The skeleton of this node is G itself (so it consists of two vertices connected by two edges).*
- ***Series case:*** *G' is not biconnected and is not a single edge. So let v_1 to v_{k-1} (with $k > 1$) be the cut vertices of G'. Since G is biconnected, we know that the blocks of G' must form a chain, so there are k blocks B_1 to B_k such that v_i is only contained in the*

two blocks B_i and B_{i+1} for $1 \leq i \leq k-1$. We assume that s is contained in B_1 and t in B_k. The graph on the left in Figure 2.4(a) shows an example for a graph where the root of the Proto-SPQR-tree is an S-node with $k = 3$. The grey areas in the figure represent biconnected subgraphs.

The root of $\mathcal{T}$ is an S-node μ where the skeleton S is a simple cycle containing the vertices $s, v_1, \ldots, v_{k-1}, t$ in that order. The edge (s,t) in S is the virtual edge *of S. If we define v_0 as s and v_k as t, then the edge $e_i = (v_{i-1}, v_i)$ represents the block B_i in G for $1 \leq i \leq k$. The graph on the right in Figure 2.4(a) shows the skeleton of the S-node for the graph on the left.*

The children of μ are defined by the graphs G_i that are constructed from the B_i by adding edge e_i for $1 \leq i \leq k$. Edge e_i is the reference edge of G_i. The i children of μ are the roots of the Proto-SPQR-trees for the subgraphs G_i.

- ***Parallel case:*** *The vertices s and t are a split pair of G' with the split components $C_1, \ldots, C_k$ where k is at least two. The graph on the left in Figure 2.4(b) shows an example for the case where the root of the Proto-SPQR-tree is a P-node with $k = 3$. The grey areas represent the split components of $\{s,t\}$. The root of $\mathcal{T}$ is a P-node μ whose skeleton S consists of the two vertices s and t and the edges $e_1, \ldots, e_{k+1}$. Edge e_i for $1 \leq i \leq k$ represents subgraph C_i while edge e_{k+1} is the virtual edge of the skeleton. The graph on the right in Figure 2.4(b) shows the skeleton of the P-node for the graph on the left.*

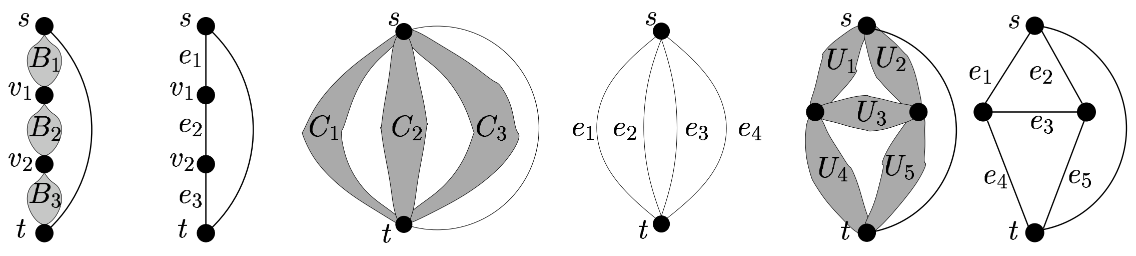
  The children of μ are defined by the graphs $G_i, \ldots, G_k$ where G_i is constructed from C_i by adding edge e_i. The children of μ are the roots of the Proto-SPQR-trees for the G_i where the reference edge is e_i.

- ***Rigid case:*** *Neither of the previous cases is applicable. So the pair $\{s,t\}$ is not a split pair of G' with at least two split components and G' is biconnected. Then the root of $\mathcal{T}$ is an R-node μ.*

  Let $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ be the maximal split pairs of G with respect to (s,t). For each of these maximal split pairs $\{s_i, t_i\}$, we define the graph U_i as the split graph of $\{s_i, t_i\}$ with respect to $\{s,t\}$. The graph on the left in Figure 2.4(c) shows an example for a graph where the root of the Proto-SPQR-tree is an R-node. The grey areas represent the split graphs of the maximal split pairs.

  The vertices in the skeleton S of μ are the vertices s and t together with all vertices s_i and t_i for $1 \leq i \leq k$. Apart from the virtual edge (s,t), S contains k edges where edge e_i connects the vertices s_i and t_i. Edge e_i represents subgraph U_i for $1 \leq i \leq k$. The graph on the right in Figure 2.4(c) shows the skeleton of the R-node for the graph on the left.

  The children of μ are defined by the graphs $G_1, \ldots, G_k$ where G_i is constructed from U_i by adding edge e_i. The children of μ are the roots of the Proto-SPQR-trees of the G_i with reference edge e_i.



(a) Case S-node (b) Case P-node (c) Case R-node

Figure 2.4: The structure of biconnected graphs and the skeleton of the root of the corresponding Proto-SPQR-tree

Now we can define the SPQR-tree of a biconnected graph using the definition of the Proto-SPQR-tree. We just add a Q-node to the Proto-SPQR-tree that represents the reference edge of the graph and make it the root of the SPQR-tree.

Definition 2.2 (SPQR-tree) *The SPQR-tree $\mathcal{T}$ of a biconnected planar graph G consists of a Q-node representing the reference edge e whose child is the root of the Proto-SPQR-tree for G with reference edge e.*

We will now give an example for the construction of an SPQR-tree. The example graph G is shown in Figure 2.5. We define the reference edge as the edge $(10, 6)$.

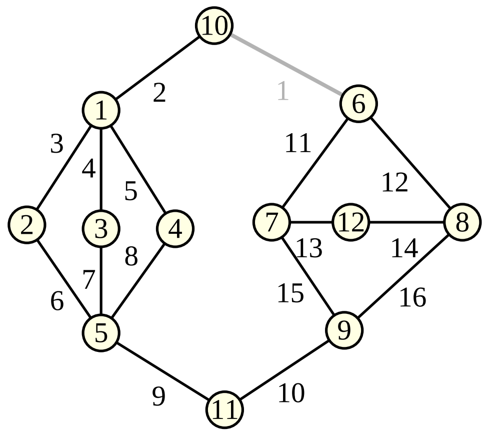


Figure 2.5: Example graph for the construction of an SPQR-tree. The grey edge is used as the reference edge.

If we delete edge $(10, 6)$, the remaining graph has cut vertices. These vertices are 9, 11, 5 and 1. So the root of the Proto-SPQR-tree will be the S-node S_1. Its skeleton is a cycle consisting of the vertices $6, 9, 11, 5, 1$ and 10 and six edges. Figure 2.6 shows the skeleton of the S-node.

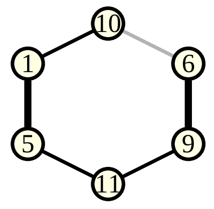


Figure 2.6: The skeleton of the root of the Proto-SPQR-tree for the graph in Figure 2.5. The grey edge is the virtual edge of the skeleton. The thick edges correspond to subgraphs of the original graph.

The children of the S-node are defined by the graphs represented by the edges in the skeleton. The edges $(9, 11), (11, 5)$ and $(1, 10)$ correspond to edges of the original graph, so the children for these edges are Q-nodes. The edges $(6, 9)$ and $(5, 1)$ represent more complex subgraphs of the original graph.

First we construct the Proto-SPQR-tree for the subgraph represented by the edge $(5, 1)$ in the skeleton of the S-node. The graph we have to decompose is shown in Figure 2.7. It was constructed by adding edge $(5, 1)$ to the subgraph of G that is represented by edge $(5, 1)$ in the skeleton of S_1. The reference edge of this graph is the edge $(5, 1)$.

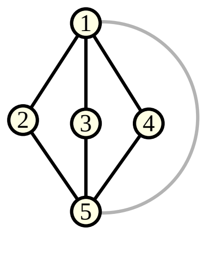


Figure 2.7: The graph that defines the subtree of the SPQR-tree defined by edge $(5, 1)$ in the skeleton of Figure 2.6. The grey edge represents the rest of the graph in Figure 2.5.

If we delete the reference edge, the split pair $\{1, 5\}$ has three split components. Therefore, the root of the Proto-SPQR-tree for this graph is a P-node P_1. The skeleton of P_1 consists of the two vertices 1 and 5 and four edges (see Figure 2.11(b)). The children of P_1 are the Proto-SPQR-trees for graphs that look like the one shown in Figure 2.8. The vertex x in the drawing is either $2, 3$ or 4. It is not hard to see that each of these graphs result in a Proto-SPQR-tree that consists of an S-node with two Q-nodes as children. The skeleton of each of the S-nodes S_2, S_3 and S_4 is isomorphic to the graph in Figure 2.8.

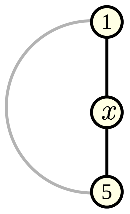


Figure 2.8: The structure of the graphs that define the children of the P-node P_1. The grey edge represents the rest of the graph in Figure 2.5.

We continue with the decomposition of the subgraph represented by edge $(6, 9)$ in the skeleton of S_1 (see Figure 2.11(a) for the skeleton of S_1). So we have to build the Proto-SPQR-tree for the graph that we get by adding edge $(6, 9)$ to the subgraph of G defined by the vertices $6, 7, 8, 9$ and 12 (this subgraph is shown in Figure 2.9). If we delete the reference edge $(6, 9)$, we see that the pair $\{6, 9\}$ is not a split pair of the graph and that it has no cut vertex. Therefore, the root of the Proto-SPQR-tree of this graph is an R-node R_1.

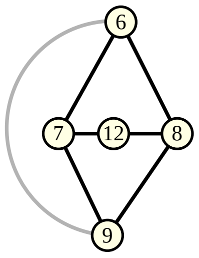


Figure 2.9: The graph represented by edge $(6, 9)$ in the skeleton of the root of the SPQR-tree. The grey edge represents the rest of the graph in Figure 2.5.

To compute the skeleton of R_1 and its children, we have to find the maximal split pairs with respect to the edge $(6, 9)$. These are the split pairs in the graph whose split graph with respect to edge $(6, 9)$ is not contained in the split graph of another split pair with respect to $(6, 9)$. The split pairs that have this property are $\{7, 6\}$, $\{7, 9\}$, $\{8, 9\}$, $\{6, 8\}$ and $\{7, 8\}$. The split pairs $\{7, 12\}$ and $\{8, 12\}$ are not maximal because their split graphs (the edge $(7, 12)$ and the edge $(8, 12)$ respectively) are contained in the split graph of $\{7, 8\}$ (the split graph of $\{7, 8\}$ consists of the two edges $(7, 12)$ and $(8, 12)$).

So the skeleton of R_1 is different from the graph in Figure 2.9 only in the fact that the two edges $(7, 12)$ and $(8, 12)$ have been replaced by a single edge $(7, 8)$ (see Figure 2.11(c) for a drawing of the skeleton of R_1). All the edges in the skeleton except $(7, 8)$ represent edges in the original graph. So four of the children of R_1 are Q-nodes. The fifth child is the root of the Proto-SPQR-tree for the simple cycle containing the three vertices $7, 12$ and 8. This Proto-SPQR-tree consists of the S-node S_5 whose two children are two Q-nodes.

The computation of the Proto-SPQR-tree of G is finished and we construct the SPQR-tree of G by introducing a Q-node for edge $(10, 6)$ and making it the root of the SPQR-tree. The resulting tree is shown in Figure 2.10. The index of each Q-node is the edge number of the edge in G that is represented by the node. These are the edge numbers shown in Figure 2.5. The labels of the inner vertices of the tree are the ones used in the description of the decomposition above. Figure 2.11 shows the skeletons of the inner nodes of the SPQR-tree.

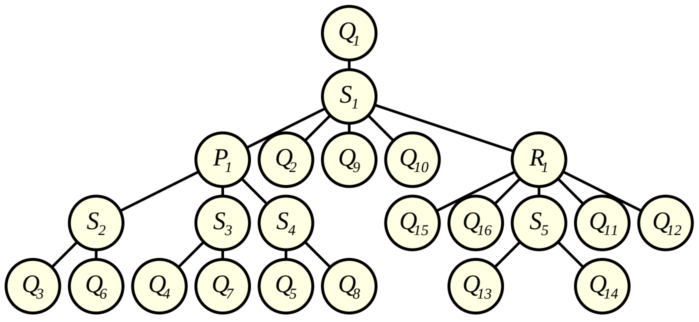


Figure 2.10: The SPQR-tree for the graph shown in Figure 2.5 with reference edge $(10, 6)$. The skeletons of the inner nodes are shown in Figure 2.11.

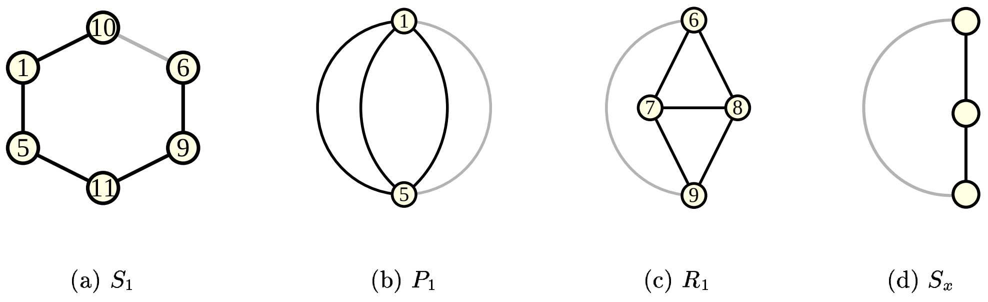


Figure 2.11: The skeletons of the inner nodes of the SPQR-tree in Figure 2.10. The grey edges are the virtual edges of the skeletons. Since the skeletons of the nodes S_2,S_3, S_4 and S_5 are isomorphic, their structure is shown in Figure 2.11(d) without vertex numbers.

2.4 Properties of SPQR-Trees

In this section, we establish some properties of SPQR-trees that play an important part in the rest of this thesis. All these properties are already mentioned in previous publications that deal with SPQR-trees (see for example (5; 6; 4) and (3)). Here, we only mention the properties that are important for our applications.

Observation 2.4 *All leaves of an SPQR-tree $\mathcal{T}$ of a graph are Q-nodes.*

This is a trivial observation since all node types except the Q-nodes are defined in such a way that they have at least two children. The only nodes in the tree that have less than two incident edges are the Q-nodes.

Observation 2.5 *The skeletons of R-nodes are triconnected graphs.*

The reason is that we replace all the maximal split pairs with single edges and add the virtual edge. Therefore, there are no more separation pairs in the skeleton (there are still split pairs since each edge forms a split pair) and so the skeleton is triconnected.

Proof Let G be a graph with reference edge $e = (s,t)$ such that the root of the Proto-SPQR-tree is an R-node μ. Let G' be the graph generated from G by deleting e. We know that G' is biconnected and that $\{s,t\}$ is not a split pair of G'. We construct the skeleton S of μ by first computing all maximal split pairs $\{s_i,t_i\}$ with respect to $\{s,t\}$. For each pair $\{s_i,t_i\}$, its split graph is replaced by a single edge. The edge $\{s,t\}$ is added to the result to produce the skeleton S of μ.

First we observe that S is not a multi graph because we replace the split graphs of each maximal split pair with a single edge. We do not produce parallel edges in the construction of S. Any parallel edges contained in G are represented by the same edge in S.

So we construct S out of G' by replacing certain subgraphs with edges and adding $\{s,t\}$. Each subgraph G_i of G we replace by an edge is the split graph of some split pair $\{s_i,t_i\}$ in G. So G_i is connected to the rest of G' only via the two vertices s_i and t_i. Replacing each G_i with an edge (s_i,t_i) does not destroy the biconnectivity of G'. Therefore, S' is also biconnected.

We want to show that S is triconnected. Since S has no multiple edges, we can use Theorem 2.1 on page 19: To show that S is triconnected, it suffices to show that S does not contain a pair of vertices whose removal splits the graphs into more than one connected component (a separation pair).

Assume that $\{v_1,v_2\}$ is a pair of vertices in S whose removal splits S into more than one connected component. Since we do not add vertices in the construction of S, these vertices must also be contained in G. The construction of S guarantees that $\{v_1,v_2\}$ is also a separation pair in G. But a separation pair is always a split pair. So $\{v_1,v_2\}$ is a split pair of G.

The definition of the skeleton of an R-node guarantees that $\{s,t\}$ is not a separation pair of G or S. Thus we can exclude that the two sets $\{v_1,v_2\}$ and $\{s,t\}$ are identical. Let G_s be the split graph of $\{v_1,v_2\}$ with respect to $\{s,t\}$. We distinguish the following two cases:

1. $\{v_1,v_2\}$ **is a maximal split pair of** G**.** In this case, the split graph G_s was replaced with a single edge in the construction of S. So $\{v_1,v_2\}$ cannot be a separation pair of S. This is a contradiction to our assumption.

2. $\{v_1,v_2\}$ **is not a maximal split pair of** G**.** Then there exists a maximal split pair sp such that the split graph of sp with respect to (s,t) properly contains the split graph of $\{v_1,v_2\}$ with respect to (s,t). So at least one of the vertices v_1 and v_2 must be contained

in the split graph of sp. It follows that not both vertices v_1 and v_2 are contained in S. This is a contradiction to our assumption.

This shows that the skeleton of an R-node in an SPQR-tree is always triconnected. $\square$

Observation 2.6 *Let G be a biconnected graph with reference edge e and $\mathcal{T}$ the corresponding SPQR-tree. If $\mathcal{T}$ has only one inner node, then the skeleton of that node is isomorphic to G.*

This observation is easy to verify by looking at the three possible types for the only inner node μ of $\mathcal{T}$. In all three cases, the graphs that are replaced by edges in the skeleton of μ are just single edges of G. Therefore, the skeleton of μ is isomorphic to G.

Observation 2.7 *Let G be a biconnected graph with at least two edges and let $\mathcal{T}$ be the Proto-SPQR-tree of G with respect to reference edge e. Then there is exactly one Q-node q_i in $\mathcal{T}$ for each edge $e_i \in E - \{e\}$ where e_i is contained in the skeleton S_i of q_i.*

When there is only one inner node in $\mathcal{T}$, it follows easily from the definition of Proto-SPQR-trees that there is one Q-node for each non-reference edge in G. If $\mathcal{T}$ has more than one inner node, it suffices to show that the graphs G_i that define the children of the root node form a partition of the non-reference edges, because by induction there will be a Q-node for each non-reference edge in each G_i. It is easy to verify that the G_i indeed define a partition of the non-reference edges by looking at the definitions of the G_i for the node types S, P and R.

A corollary of observation 2.7 is that there is exactly one Q-node for each edge of a biconnected graph in the SPQR-tree of the graph, independent of the choice of the reference edge. The reason is that we explicitly create a Q-node for the reference edge when we construct the SPQR-tree from the Proto-SPQR-tree.

The next observation shows that the choice of the reference edge is not important. The reference edge defines only which Q-node is the root of the tree. If we think of the SPQR-tree as an unrooted tree, the choice of the reference edge has no impact whatsoever.

Observation 2.8 *Let G be a biconnected graph and e_1 and e_2 two of its edges. Let $\mathcal{T}_1$ be the SPQR-tree of G with reference edge e_1 and $\mathcal{T}_2$ the SPQR-tree of G with reference edge e_2. Modifying $\mathcal{T}_1$ by making the Q-node that represents e_2 the root of the tree results in $\mathcal{T}_2$.*

So if we think of the SPQR-tree as an acyclic connected graph instead of a rooted tree, then there is just one SPQR-tree for any biconnected graph. The SPQR-tree is a decomposition of a graph that is determined by the split pairs of a graph. The reference edge is only the point where we start the decomposition. The result of this decomposition will always be the same, no matter which edge we choose as the reference edge.

Lemma 2.1 *The size of the data structure SPQR-tree for a biconnected graph $G = (V, E)$ is linear in the size of E.*

Proof We have shown that the leaves of the SPQR-tree are in one-to-one correspondence with the edges in E. Since every inner node of the tree has at least two children, the total number of nodes in the tree must grow linearly with the number of edges in G.

If a node of the tree has k children, there can be at most $k+1$ edges in its skeleton, because each edge except the virtual edge is associated with a child of the node. Thus, the sum of the number of edges in all skeletons is bounded from above by a constant multiple of the number of edges in the tree. So the number of all edges in all skeletons grows also linearly with the number of edges in the graph.

For all four types of nodes we know that the number of vertices in the skeleton are either constant (Q- and P-nodes) or at most the number of edges (S- and R-node skeletons are connected graphs without parallel edges). With the same argument used for the edges we can show that the sum of the number of vertices in all skeletons grows linear with the number of edges in the graph. $\square$

Corollary 2.1 *The size of the data structure SPQR-tree for a planar graph G is linear in the number of vertices of G if G has no parallel edges.*

This follows from Lemma 2.1 because a planar graph without parallel edges can have at most $3n - 6$ edges.

Theorem 2.2 *Let $G = (V, E)$ be a biconnected graph. Then the SPQR-tree of G can be computed in time $O(|E|)$.*

The proof relies on the linear time algorithm for dividing a graph into triconnected components by Hopcroft and Tarjan (32). Gutwenger and Mutzel corrected some mistakes in this algorithm and presented the first linear time implementation of an algorithm for computing the SPQR-tree for a biconnected graph (27). Their paper contains pseudo code and a proof of the running time. The code is contained in the AGD-library (AGD).

2.5 SPQR-Trees and Combinatorial Embeddings

As already mentioned in Section 2.1, SPQR-trees can be used to enumerate all embeddings of a planar biconnected graph. In this section, we want to show how this can be done. First, we have to define the embeddings of a planar graph.

Definition 2.3 (Weakly Equivalent Drawings) *Two planar drawings D_1 and D_2 of a graph G are* weakly equivalent*, if for every vertex v in G, the circular order of the incident edges around v in clockwise order is the same in D_1 and D_2.*

This is a statement about the topology of a drawing, it does not make any statements about the length and shape of edges or the size and absolute position of vertices. Two drawings can look very different and still be weakly equivalent. On the other hand, two drawings can look very similar and at the same time not be weakly equivalent.

Consider the four drawings in Figure 2.12. They show four different drawings of the same planar graph. The two drawings 2.12(a) and 2.12(c) are weakly equivalent. The two drawings look quite different, but the sequence of the edges in clockwise order around each vertex is the same. This is also true for the two drawings 2.12(b) and 2.12(d).

The two drawings 2.12(a) and 2.12(d) look very similar but are not weakly equivalent. The sequences of the edges around each vertex in clockwise order are mirrored in the two drawings. In fact, the two drawings are mirror images of each other.

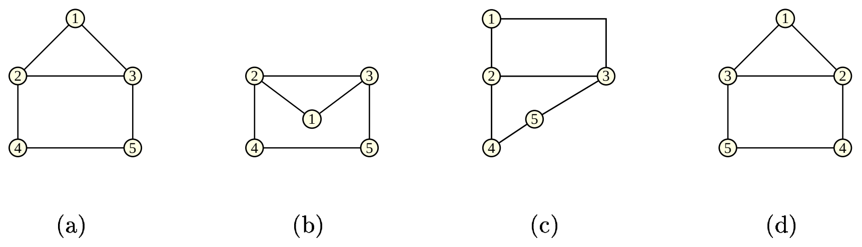


Figure 2.12: Four different drawings of the same planar graph

Definition 2.4 (Combinatorial Embeddings) *The combinatorial embeddings of a planar graph G are the equivalence classes of its planar drawings with respect to the weak equivalence relation.*

We often say a drawing *realizes* a combinatorial embedding Π if it belongs to the equivalence class Π.

To define planar embeddings, we first have to introduce the concept of the *faces* of a planar drawing. A planar drawing partitions the plane into connected regions. These regions are called *faces*. The unbounded face in the drawing is called the *external face*. The *boundary* of a face f is the subgraph of G, whose image in the drawing forms the boundary of f. If we consider Figure 2.12(a), then the boundary of the triangular face consists of the vertices 1, 2, and 3 together with the edges connecting these vertices.

Definition 2.5 (Strongly Equivalent Drawings) *Two drawings D_1 and D_2 are strongly equivalent if they are weakly equivalent and the boundary of the external face is the same in both drawings.*

We use the strong equivalence relation for planar drawings to define planar embeddings.

Definition 2.6 (Planar Embedding) *The planar embeddings of a graph are the equivalence classes of its drawings with respect to the strong equivalence relation.*

The following theorem states the most important point in this section: SPQR-trees can be used to represent every combinatorial embedding of a biconnected planar graph. This is the main feature of SPQR-trees and has been described in most of the publications that use them.

Theorem 2.3 *Let G be a biconnected planar Graph and let $\mathcal{T}$ be its SPQR-tree. A combinatorial embedding Π for G uniquely defines a combinatorial embedding of the skeleton of each node in $\mathcal{T}$. On the other hand, fixing the combinatorial embedding of the skeleton of each node in $\mathcal{T}$ uniquely defines a combinatorial embedding of G.*

Intuitively, it is very easy to see why the theorem holds. Imagine a planar drawing of the original graph G. By replacing the drawings of certain subgraphs by the drawings of a single edge we can obtain a planar drawing for any skeleton in the SPQR-tree of G. So it is not hard to see that a combinatorial embedding of G defines a combinatorial embedding of each skeleton.

Now assume that we have fixed a combinatorial embedding of each skeleton. We can always merge two skeletons whose nodes are adjacent in the SPQR-tree by identifying the common virtual edge in both skeletons and then deleting it. Thus we can turn two nodes of the SPQR-tree into a single node. We can continue with this operation until there is only one node left in the tree. The skeleton of this node is isomorphic to the original graph. It is not hard to see that this merging operation can be used to construct a combinatorial embedding of the original graph from the combinatorial embeddings of the skeletons.

Chapter 3

Inserting an Edge into a Planar Graph with the Minimum Number of Crossings

This chapter deals with the problem of inserting an additional edge into a planar graph under the condition, that the edges of the original graph do not cross each other in the resulting drawing. The goal is to minimize the number of crossings of the new edge with the original edges. Alternatively, the problem can be stated as finding a combinatorial embedding of a planar graph in which the given edge can be inserted with the minimum number of crossings. Many problems concerned with the optimization over the set of all combinatorial embeddings of a planar graph have been shown to be NP-hard. We found a conceptually simple linear time algorithm based on SPQR-trees, which is able to find a crossing minimum solution. This result was first published in (28).

3.1 Introduction to the Problem

Crossing minimization is among the most challenging problems in graph theory and graph drawing. Although there is a vast amount of literature on the problem, so far no practically efficient exact algorithm for crossing minimization is known even for small instances. Currently, one of the most popular approaches for crossing minimization is based on planarization. In the first step of the method, a small number of edges is deleted so that the resulting graph is planar. Then, the edges are iteratively re-inserted into the planar subgraph in such a way, that the number of crossings produced is small.

The last step of the planarization method is usually done in the following way: Fix an arbitrary combinatorial embedding Π of the planar subgraph P and re-insert the first deleted edge e_1. Edge e_1 is re-inserted by solving a shortest path problem in the augmented (geometrical) dual graph of P associated with Π. Every crossing of e_1 with an edge f corresponds to using an edge in the dual graph. Replacing the generated crossings by artificial vertices yields a planar graph again. Now, the next edge can be inserted, and so on.

One criticism of the planarization method is that when choosing a “bad” embedding in the edge re-insertion phase, the number of crossings may get much higher than necessary (29). Hence, the question arises whether there is a polynomial time algorithm for inserting an edge into the planar graph P so that the number of generated crossings is minimized. Therefore, the task is to optimize over the set of all possible combinatorial embeddings of P.

While it is possible to compute an arbitrary combinatorial embedding for a planar graph in linear time (16; 41), it is often hard to optimize over the set of all possible combinatorial embeddings. E.g., the problem of bend minimization can be solved in polynomial time for a fixed combinatorial embedding (47), while it is NP-hard over the set of all combinatorial embeddings (24). When a linear function of polynomial size is defined on the cycles of a graph, it is NP-hard to find the embedding that maximizes the value of the cycles that are face cycles in the embedding (43; 42). A sketch of a proof for this fact can also be found in Section 4.2.4 on page 74. Woeginger has shown that it is in general NP-complete to decide if there is a combinatorial embedding of a graph with at least k face cycles of length at least five (50).

This chapter shows that the edge re-insertion problem can be solved in linear time, thus solving a long standing open problem in graph drawing. We present a conceptually simple linear time algorithm based on SPQR-trees which is able to solve the edge re-insertion problem to optimality.

Figure 3.1 shows a simple case where the choice of the combinatorial embedding of the planar subgraph has an impact on the number of crossings produced when inserting the dashed edge. When choosing the embedding of Figure 3.1(a) for the planar subgraph (without the dashed edge), we get two crossings, while the optimal crossing number over the set of all combinatorial embeddings is one (as shown in Figure 3.1(b)).

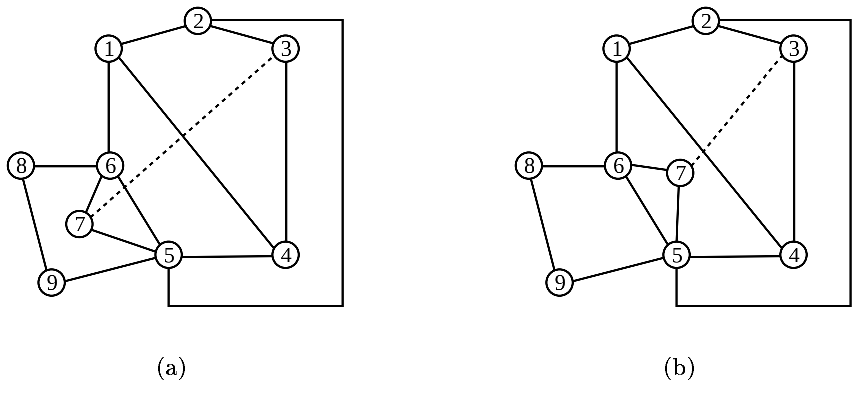


Figure 3.1: The number of crossings when inserting an edge highly depends on the chosen embedding

Formally, we define the edge insertion problem as follows: Given a planar graph $G = (V, E)$ and a pair of vertices (v_1, v_2) in G, find a combinatorial embedding Π of G such that we can add the edge $e = (v_1, v_2)$ to Π with the minimum possible number of crossings among all combinatorial embeddings of G. We will present an algorithm that is able to solve the problem in linear time.

Note that the solution produced by our algorithm does not necessarily lead to a drawing of the graph $G' = (V, E \cup \{e\})$ with the minimum number of crossings. The reason is that there may not always be a drawing with the minimum number of crossings where the edges of G do not cross each other.

In section 3.2, we define some necessary concepts and introduce the traversing cost of an edge in the skeleton of an SPQR-tree. Section 3.3 contains the algorithm for solving our problem for planar biconnected graphs. We also prove the correctness in this section and discuss the running time. In section 3.4 we present a generalization of the algorithm for arbitrary planar graphs. Again we proof the correctness of the algorithm and give the running time. Section 3.5 gives our computational results on a set of benchmark graphs. The last section addresses the difference between finding a drawing with the minimum number of crossings and the problem discussed in this chapter.

3.2 Preliminaries

In this chapter, we describe an algorithm for solving the following problem: Given a planar graph G and two non-adjacent vertices, find an optimal edge insertion path and a corresponding combinatorial embedding Π of G. So first we have to define what is meant by the term edge insertion path.

In this definition, we need the term *dual graph*.

Definition 3.1 (Dual Graph) *Let G be a planar graph and Π a combinatorial embedding of G. Then the* dual graph $\overline{G_\Pi} = (F, \bar{E})$ *of G with respect to Π is defined as follows:*

1. *The vertex set F corresponds to the set of faces in Π.*

2. *There is a one to one correspondence between the edges of G and the edges of $\overline{G_\Pi}$. Let e be an edge of G and $\bar{e}$ the corresponding edge in $\bar{E}$. Then $\bar{e}$ connects the vertices of $\overline{G_\Pi}$ that correspond to the faces of Π that have e on their boundary.*

Figure 3.2 shows an example of a planar graph and its dual graph. The vertices drawn as squares and the edges drawn as straight lines represent the primal graph while the vertices drawn as circles and the curved edges represent the dual graph. The dual vertices are drawn into the faces that they represent. Each dual edge is drawn in such a way that it intersects the corresponding edge of the primal graph.

Note that dual graphs are often multi graphs, since several edges can be on the boundary of the same faces in the embedding of a planar graph. If G is not biconnected, the dual graph of G may also contain self loops (as in Figure 3.2). This is the case if there exists an edge in G that is on the boundary of only one face. The dual graph of a tree for example contains only one vertex and a self loop for every edge in the tree because every embedding of a tree has only one face.

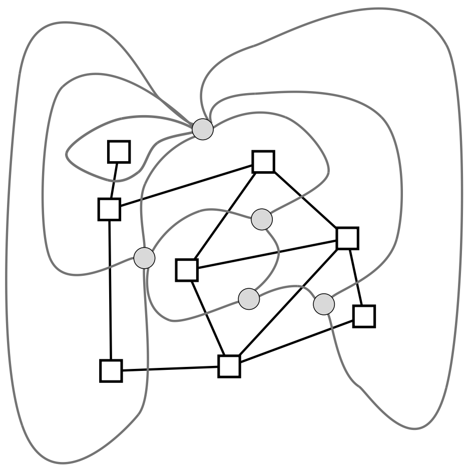


Figure 3.2: A planar graph and its dual graph

Now we can define edge insertion paths. Informally, an edge insertion path is a list of edges with the property that we can insert an additional edge into a planar graph such that the new edge crosses only the edges in the list in the sequence given by the list.

Definition 3.2 *Let v_1 and v_2 be two non-adjacent vertices in a planar graph G and Π a combinatorial embedding of G. Let $\overline{G_\Pi}$ be the dual graph of G with respect to Π. By $\bar{e}$, we denote the edge in $\overline{G_\Pi}$ corresponding to edge e in G. With f we denote the vertex in $\overline{G_\Pi}$ corresponding to face f in Π. Then the list $L = (e_1, \ldots, e_k)$ of edges in G is an* edge insertion path *for v_1 and v_2 in G with respect to Π if either L is empty and v_1 and v_2 are on the boundary of the same face in Π or the following three conditions are satisfied:*

1. *There is a face in Π with e_1 and v_1 on its boundary.*

2. *There is a face in Π with e_k and v_2 on its boundary.*

3. *$\bar{L} = (\overline{e_1}, \ldots, \overline{e_k})$ is a path in $\overline{G_\Pi}$.*

This basically means that we can add the edge (v_1, v_2) to Π with k crossings, where the i-th crossing involves edge (v_1, v_2) and edge e_i for $1 \leq i \leq k$. We call an edge insertion path an *optimal edge insertion path* for v_1 and v_2, when there is no shorter edge insertion path for v_1 and v_2 with respect to any combinatorial embedding of the graph.

Figure 3.3 shows three different edge insertion paths for v_1 and v_2 with respect to the embedding realized by the drawing. The edge insertion paths are represented by grey lines that connect v_1 and v_2 and cross the edges belonging to the edge insertion path that they represent. The three paths are the empty path, the path (e_1, e_2, e_3) and the path (e_4, e_5, e_6). In this case, the empty path is the only optimal edge insertion path for v_1 and v_2.

One term that we will need in the description of our algorithm is the *expansion graph* of an edge $e = (u, v)$ in a skeleton. Intuitively, we can think of the expansion graph of an edge

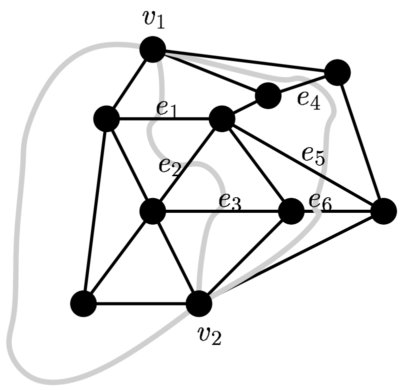


Figure 3.3: Three different edge insertion paths for v_1 and v_2

in the skeleton as the subgraph of G that is represented by that edge together with e. Each skeleton is a simplified representation of the original graph G and defines a decomposition of G into split components. Each edge in the skeleton represents one of these split components.

Definition 3.3 (Expansion Graph) *Let $e = (u, v)$ be an edge in a skeleton S of a node μ of the SPQR-tree $\mathcal{T}$ of G. Since e is an edge in a skeleton, the vertices u and v are a split-pair of G. Then we define the* expansion graph *of e as follows:*

1. *If μ is the Q-node for edge e' in G, then the expansion graph $G(e)$ of e is defined in as follows: if e is the virtual edge of μ then we define $G(e)$ as $G(e) = (V, (E - \{e'\}) \cup \{e\})$. Otherwise, we define $G(e)$ as $G(e) = (\{u, v\}, \{e, e'\})$. So the expansion graph is either isomorphic to G or to S.*

2. *If μ is an R-node or S-node, then the expansion graph of e is the union of all the split components of the split pair $\{u, v\}$ in G that contain no vertices of S except u and v together with edge e.*

3. *If μ is a P-node, then there are at least three split components of the pair $\{u, v\}$ in G. In the construction of $\mathcal{T}$, all edges e_i of S except the virtual edge were associated with a subgraph G_i of G. So we define the expansion graph of each e_i as the graph G_i together with edge e. The expansion graph of the virtual edge is defined as the split component of $\{u, v\}$ that contains the reference edge of $\mathcal{T}$ (the edge of G that was used to start the decomposition) together with edge e.*

The definition of the expansion graph is similar to our definition of the split graph of an edge in a skeleton. The difference is that the split graph is defined with respect to an edge in G while the expansion graph is defined by the structure of the SPQR-tree. Figure 3.4 shows an example for this concept. Assume the graph on the right is a P-nodes skeleton in the SPQR-tree of the graph on the left. So the three edges in the skeleton correspond to the three split components of the split pair $(0, 1)$ in the graph on the left. The expansion graph of the grey edge on the right is the subgraph of the graph on the left shown in grey.

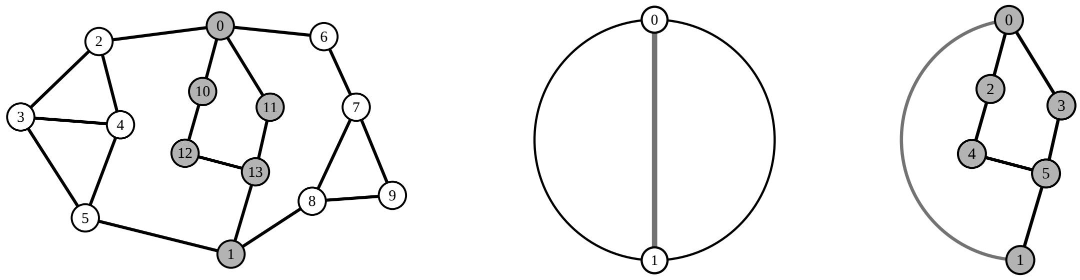


Figure 3.4: In the SPQR-tree for the graph on the left, the expansion graph of the grey edge in the P-node skeleton in the middle is the graph on the right.

Now we need to define the *traversing cost* of the expansion graph of an edge in the skeleton of a node in an SPQR-tree. Let e be an edge in a skeleton and $G(e)$ the expansion graph of e. Note that $G(e)$ also contains the edge e as the virtual edge of the graph. Since $G(e)$ is biconnected, any edge of the graph is on the boundary of exactly two faces in every embedding of $G(e)$.

Let $\overline{G(e)}_\Pi$ be the dual graph of $G(e)$ with respect to an embedding Π of $G(e)$ and v_1 and v_2 the vertices in $\overline{G(e)}_\Pi$ representing the two faces separated by e. We define $P(\overline{G(e)}_\Pi, e)$ as the shortest path in $\overline{G(e)}_\Pi$ that connects v_1 and v_2 and does not use edge $\bar{e}$ (the representative of edge e in $\overline{G(e)}_\Pi$). In the proof for Theorem 3.1 we will show that the length of this path is the same for every embedding of $G(e)$ and so we do not have to denote the embedding for which the shortest path was computed. Thus we define the *traversing cost* $T(e)$ of e in $G(e)$ as follows:

$$T(e) = |P(\overline{G(e)}, e)|$$

Theorem 3.1 *For every inner node μ in the SPQR-tree $\mathcal{T}$ and for every edge e in the skeleton S of μ, the traversing costs of $G(e)$ are independent of the embedding of $G(e)$.*

Proof In this proof, we denote by $\hat{e}$ the edge in $\mathcal{T}$ that corresponds to edge e in the skeleton of a node in $\mathcal{T}$. Edge $\hat{e}$ connects the two nodes of $\mathcal{T}$ that both contain e in their skeletons. We prove the theorem using induction over the length of the *longest* path P in the SPQR-tree from node μ via edge $\hat{e}$ to a Q-node of $\mathcal{T}$. Note that this length is at least one since μ is an inner node of $\mathcal{T}$.

- **$k = 1$:** The node μ' in the SPQR-tree that is connected to μ via edge $\hat{e}$ is a Q-node. The edge e corresponds to an edge e' which is contained in the graph represented by $\mathcal{T}$. The expansion graph of e consists of the two edges e and e' together with the two incident vertices. Therefore, the traversing costs of e are one.

- **$k > 1$:** We assume that the theorem is correct for the case that the longest path from μ to a Q-node using edge $\hat{e}$ has at most length n. We want to prove the correctness for

$k = n + 1$. Let μ' be the node connected to μ via edge $\hat{e}$. Then μ' is not a Q-node, otherwise the length of the longest path from μ to a Q-node using edge $\hat{e}$ would be one. So μ' is an S-node, a P-node or an R-node.

The traversing costs for all edges in the skeleton S of μ' are independent of the embedding of $G(e)$ (the expansion graph of e) because for each of these edges e'', the length of the longest path from μ' to a Q-node is at most n and by induction we know that the traversing costs of e'' are independent of the embeddings of $G(e'')$.

Let $G(e)$ be the expansion graph of e in the skeleton of μ. Let $\mathcal{D}(G(e))$ be the set of all dual graphs of $G(e)$ with respect to all possible combinatorial embeddings of $G(e)$. We are looking for a shortest path among all the paths in the graphs $\mathcal{D}(G(e))$ connecting the face left of e with the face right of e that does not use edge $\bar{e}$. Instead of looking at the dual graphs in $\mathcal{D}(G(e))$, we might as well look at the dual graphs of the skeleton S' of μ' where we give the edges costs corresponding to the traversing costs of each edge. This is true because the traversing costs of each edge e'' in S' are independent of the embedding of $G(e'')$ by induction.

Let D be the dual graph of S' for an arbitrary embedding. We will show now how to compute the shortest path in D connecting the faces left and right of e and that the length of this path does not depend on the embedding chosen for S'.

We prove the claim by arguing over the three possible node types of node μ'. Let $e_1, \ldots, e_m$ and e (the virtual edge) be the edges in the skeleton S' of μ'.

- μ' **is an** S**-node:** There is only one embedding of S' and the corresponding dual graph D consists of two vertices v_1 and v_2 connected by the dual edges $\overline{e_1}, \ldots, \overline{e_m}$ and $\bar{e}$. The shortest path from v_1 to v_2 that does not use $\bar{e}$ consists obviously of the edge $e_{min} \in \{e_1, \ldots, e_m\}$ with the smallest traversing costs. The costs of e_{min} are the traversing costs of e.
- μ' **is a** P**-node:** Independent of the choice of the embedding of S', the dual graph D of S' is a cycle with $m + 1$ edges. Depending on the embedding, the sequence of the edges in this cycle changes. This has obviously no influence on the shortest path in D between the vertices connected by $\bar{e}$ that does not use $\bar{e}$. We always have to use all edges except $\bar{e}$. Thus, the traversing costs can be computed as the sum of the traversing costs of the edges e_1 to e_m.
- μ' **is an** R**-node:** The skeleton S of μ' is triconnected and has therefore only two different combinatorial embeddings D_1 and D_2 which are mirror images of each other. Let $d_1(e)$ be the edge in D_1 corresponding to edge e in S and $d_2(e)$ the corresponding edge in D_2. The function that maps $d_1(e)$ to $d_2(e)$ for each edge e in S is a graph isomorphism from D_1 to D_2. So when we give the edges in the dual graph the costs defined by the traversing costs of the primal edges in S', we will get the same shortest path from the face left of e to the face right of e in both dual graphs.

□

So the traversing costs of an edge e in a skeleton are independent of the embedding chosen for its expansion graph $G(e)$. This suggests a simple recursive algorithm for computing the traversing costs of e:

Choose an arbitrary embedding Π of S' and compute the shortest path in the corresponding dual graph with respect to the edge weights given by the traversing costs of each edge in S'. This algorithm is correct because the traversing costs of an edge in a skeleton are independent of the embedding of its expansion graph.

Another term that we use in this chapter is *face cycle*. In a planar embedding Π of a graph G, a face cycle is a directed cycle such that in every drawing that realizes Π, the region to the right of all edges in the cycle does not contain any images of vertices or edges of G. So the face cycles are the cycles of the graph that form the boundaries of the faces in an embedding.

In the next section, we will present an algorithm for computing an optimal edge insertion path for a pair of non-adjacent vertices in a biconnected graph.

3.3 Solving the Problem for Biconnected Graphs

If a graph is biconnected, we can compute its SPQR-tree. So our new algorithm can work directly on this tree to find a shortest edge insertion path for a pair of vertices. The resulting algorithm is conceptually simple but the proof of correctness is rather complex.

First we will describe our algorithm. Then we will present the proof of correctness and finally we will show that the running time of our algorithm is linear in the number of vertices of the problem graph.

3.3.1 The Algorithm

Definition 3.4 *Let v_1 and v_2 be two vertices in the planar graph G and Π be an embedding of G. Let $\overline{G_\Pi}$ be the dual graph of G with respect to embedding Π. We say the graph $\overline{G_\Pi}'$ has been obtained by* augmenting $\overline{G_\Pi}$ with v_1 and v_2, *if it was constructed by adding v_1 and v_2 to $\overline{G_\Pi}$ and inserting edges from v_i to all vertices in $\overline{G_\Pi}$ representing faces in Π with v_i on their boundary for $i \in \{1, 2\}$.*

Figure 3.5 shows a graph and its augmented dual graph augmented with v_1 and v_2. The original graph consists of the black edges and the square vertices. The dual graph is drawn with grey edges and round vertices. The edges that were added to the dual graph to augment it with v_1 and v_2 are drawn as dashed lines.

Algorithm 1 on page 42 computes an optimal edge insertion path for a biconnected planar graph and two non-adjacent vertices in this graph. The term *allocation node* of a vertex v in a graph G used in the algorithm refers to a node in the SPQR-tree of G with v in its skeleton. We say an edge e in a skeleton *represents* a vertex v of G if v is contained in its expansion graph.

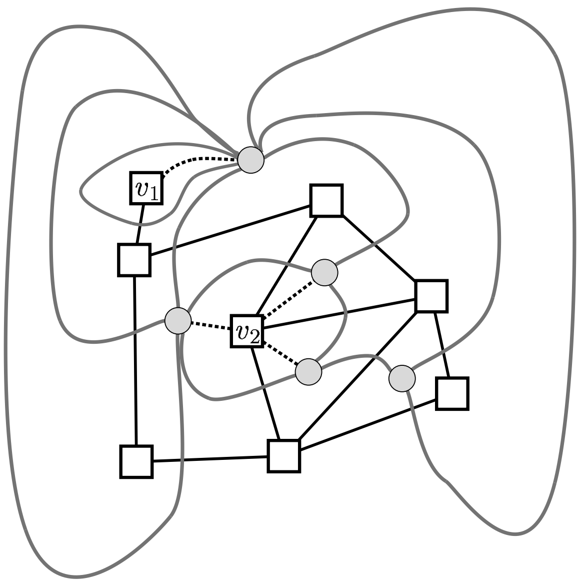


Figure 3.5: A graph (drawn with black edges and square vertices), its dual graph (round vertices and grey edges) and the additional edges added to augment the dual graph with v_1 and v_2 (dashed edges)

In one of the steps of the algorithm, edges in the skeleton of a node of the SPQR-tree are *expanded*. This operation works as follows: Let e be an edge in the skeleton S and $G(e)$ its expansion graph. We expand e by replacing it with the graph $G(e)'$ obtained from $G(e)$ by deleting its virtual edge e.

3.3.2 Correctness

The proof of correctness has two parts. The first part consists of Theorem 3.2 and shows that the list computed by the algorithm is an edge insertion path. In the second part, we show that the edge insertion path computed by the algorithm is optimal (Theorem 3.3). The first proof is quite involved while the proof of the second part is much shorter.

Theorem 3.2 *If Algorithm 1 is given a planar biconnected graph G and two non-adjacent vertices v_1 and v_2 in G, then it computes a list of edges L that represents an edge insertion path for v_1 and v_2 with respect to some embedding Π of G.*

Proof The proof centers on the nodes of $\mathcal{T}$ on path P_2. The reason for not using P_3 (the list of nodes that the algorithm works on) is that the task of the algorithm is to compute the edge insertion path L, while in this proof, we have to construct an embedding Π and show that the edge insertion path computed by the algorithm is valid for the Π. To compute the edge insertion path, we do not have to consider all the skeletons in $\mathcal{T}$ but if we want to define an embedding of G, we have to define an embedding for every skeleton.

Algorithm 1: Algorithm `OptimalBlockInserter` for computing an optimal edge insertion path for a pair of non-adjacent vertices in a biconnected planar graph

```
Input: A biconnected planar graph $G$, and two non-adjacent vertices $v_1$ and $v_2$ in $G$.
Result: An edge insertion path $L$ for $v_1$ and $v_2$ with respect to some embedding $\Pi$ of $G$
begin
    Compute the SPQR-tree $\mathcal{T}$ of $G$;
    $L \leftarrow ()$;
    Find allocation nodes $\mu_1$ of $v_1$ and $\mu_2$ of $v_2$;
    Find the path $P_1$ in $\mathcal{T}$ starting at $\mu_1$ and ending at $\mu_2$;
    Delete nodes from the start and end of $P_1$ until we have produced the shortest path
    $P_2$ in $\mathcal{T}$ from an allocation node of $v_1$ to an allocation node of $v_2$;
    Delete all nodes from $P_2$ except the $R$-nodes, producing the list $P_3$ of nodes in $\mathcal{T}$;
    while $P_3$ is not empty do
        Pop the first node $\mu$ from $P_3$ and let $S$ be its skeleton;
        if $v_1$ is in $S$ then
            $x_1 \leftarrow v_1$;
        else
            Split the edge representing $v_1$ in $S$ by inserting a new vertex $y_1$ into the edge.;
            Mark the two edges produced by the split;
            $x_1 \leftarrow y_1$;
        if $v_2$ is in $S$ then
            $x_2 \leftarrow v_2$;
        else
            Split the edge representing $v_2$ in $S$ by inserting a new vertex $y_2$ into the edge;
            Mark the two edges produced by the split;
            $x_2 \leftarrow y_2$;
        Expand all unmarked edges in $S$;
        Compute an arbitrary embedding $\Pi$ for $S$;
        Compute the dual graph $\overline{S_\Pi}$;
        Augment $\overline{S_\Pi}$ with $x_1$ and $x_2$;
        Compute the shortest path $L'$ in $\overline{S_\Pi}$ from $x_1$ to $x_2$;
        Delete the first and the last edge in $L'$;
        Replace every dual edge in $L'$ by its primal counterpart;
        Append $L'$ to $L$;
end
```

Let P_2 be given by the sequence $P_2 = (p_1, \ldots, p_k)$ of nodes in $\mathcal{T}$. The node p_1 is an allocation node of v_1 and p_k is an allocation node of v_2. In other words: the skeleton of p_1 contains v_1 and the skeleton of p_k contains v_2 (note that p_1 and p_k might be identical). By the construction of P_2, none of the nodes p_i with $1 < i < k$ has the vertex v_1 or v_2 in its skeleton.

We call these nodes the *middle nodes* of P_2.

We construct an embedding Π of G iteratively. The construction is done in k stages. We build graphs G_i for $i \in \{1, \dots, k\}$ and for each of these graphs we construct an embedding Π_i. The graph G_k is G and the embedding Π_k is an embedding of G such that L is an edge insertion path for v_1 and v_2 with respect to Π_k. Each graph G_i for $1 \leq i < k$ can be constructed from G_{i+1} by replacing a subgraph with a path of two edges. The inner vertex of this path is called r_i.

We will prove that for all $i \in \{1, \dots, k-1\}$, the prefix L_i of L with the property that all edges of L_i are contained in G_i is an edge insertion path for v_1 and r_i with respect to Π_i and that the suffix of L following L_i does not contain any edges from G_i. Each L_i with $i \in \{1, \dots, k-1\}$ is a prefix of L_{i+1} (see Figure 3.6).

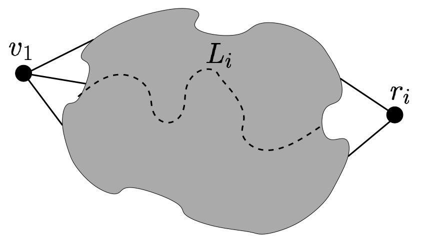


Figure 3.6: The graph G_i of stage i in the proof for Theorem 3.2

We first look at the special case where there is only one node p on P_2. This means that v_1 and v_2 are both contained in the skeleton S of p. We consider three sub-cases:

1. **p is an S-node:** Then v_1 and v_2 must be a separation pair of G as shown in Figure 3.7. A separation pair of a planar graph can always be connected by an edge without producing crossings independent of the chosen embedding. The list L computed by our algorithm is empty. Thus, L is an edge insertion path for v_1 and v_2 with respect to any embedding of G, and our algorithm calculates the correct result.

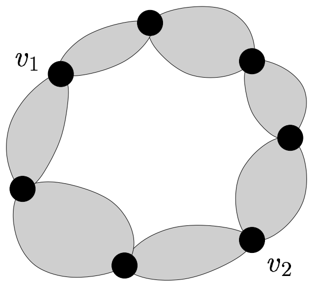


Figure 3.7: Example for the structure of G in the case where P_2 consists of a single S-node. The grey areas represent biconnected subgraphs.

2. p **is a** P**-node:** Again, v_1 and v_2 are a separation pair of G and we can use the same argument as in the first case (see Figure 3.8).

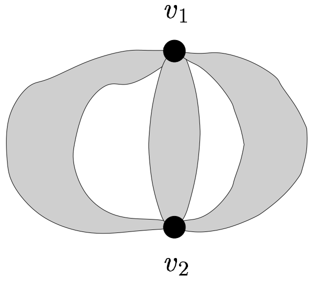


Figure 3.8: Example for the structure of G in the case where P_2 consists of a single P-node

3. p **is an** R**-node:** In this case, the expansion done in our algorithm produces the original graph G, because all edges are expanded. Therefore, the algorithm computes an edge insertion path in G for v_1 and v_2 with respect to the embedding Π computed in the algorithm.

Now we will show the following fact for the case that P_2 has more than one node: Let G_i for $i \in \{1, \ldots, k-1\}$ be the graph that is constructed by expanding all edges in the skeleton of p_i except the representative $e_2 = (u_1, u_2)$ of v_2 and then replacing e_2 with a path of length two linked by the new vertex r_i. Then there is an embedding Π_i of G_i such that the prefix L_i of L containing only edges in G_i is an edge insertion path for v_1 and r_i with respect to Π_i (see Figure 3.6). The suffix of L following L_i does not contain any edges from G_i.

The last fact is true because our algorithm works through P_3 (a subsequence of P_2) sequentially. In each execution of the loop of the algorithm, we consider a graph S', constructed from the skeleton of an R-node p_i on P_3 by expanding all edges of its skeleton except the representatives of v_1 and v_2.

The list L' appended to L after each loop execution consists only of edges in S'. But the graphs S' computed in each iteration of the **while** loop are edge-disjoint. The reason for this is that we do not expand the representative of v_1 (so we cannot see the edges treated in previous iterations) and we do not expand the representative of x_2 (so we cannot see the edges treated in future iterations).

We show by induction over i that L_i is an edge insertion path for v_1 and r_i in G_i with respect to some embedding Π_i of G_i.

1. $i = 1$: We consider four cases for p_1:

   (a) p_1 **is a** Q**-node:** This cannot happen since all Q-nodes are leaves of $\mathcal{T}$ and the adjacent node in $\mathcal{T}$ also has v_1 in its skeleton. In the step from P_1 to P_2 in the algorithm, we have discarded the node.

   (b) p_1 **is a** P**-node:** This cannot happen, because all nodes in $\mathcal{T}$ adjacent to p_1 have v_1 in their skeleton. Thus, we have deleted p_1 in the step from P_1 to P_2.

(c) p_1 **is an** S**-node:** In this case, v_1 forms a separation pair together with any of the vertices adjacent to r_1. This is the case because all these vertices are contained in the skeleton of the S-node. It follows that v_1 and r_i are a separation pair of G_1 (see Figure 3.9(a)).

Thus, we can insert the edge (v_1, r_1) without any crossings independent of the embedding of G_1. So Π_1 is just an arbitrary embedding of G_1 and the prefix of L that is an edge insertion path for v_1 and r_1 with respect to Π_1 is the empty list of edges. Therefore, we define L_1 as the empty list of edges.

(d) p_1 **is an** R**-node:** Our algorithm constructs G_1 (note that r_1 is called y_2 in the algorithm), computes an arbitrary embedding Π_1 for G_1 and an edge insertion path L' for v_1 and r_1 with respect to Π_1 (see Figure 3.9(b)). L' is the prefix of L where all the edges are contained in G_1. We define L_1 as L'.

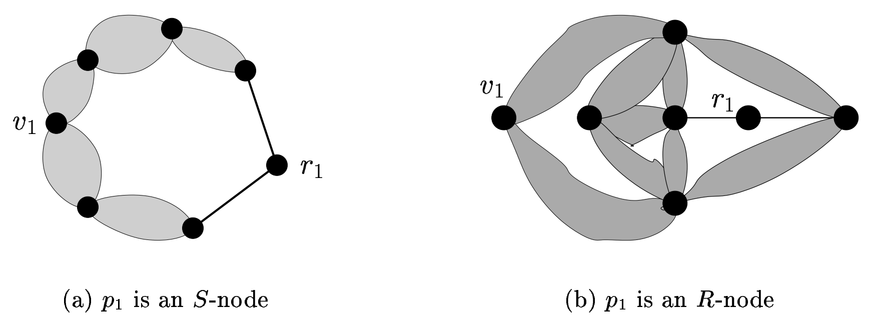


Figure 3.9: Examples for the structures of graph G_1 in the cases where p_1 is either an S-node or an R-node

2. $1 < i < k$: We assume $i = l$ and we know that our assumption is correct for all $i < l$. Let x_1 and x_2 be the representatives of v_1 and v_2 in the skeleton S_l of p_l. We now look at the graph G'_l that we get by expanding all edges in S_l except for x_1 and x_2 and by splitting x_j for $j \in \{1, 2\}$ introducing the vertices r_{l-1} and r_l (r_{l-1} splits x_1 and r_l splits x_2). We determine an embedding Π'_l for G'_l as follows, depending on the type of node p_l:

   (a) p_l **is a** Q**-node:** Again, this cannot happen for the same reason as in the induction basis.

   (b) p_l **is an** S**-node:** In this case, we choose an arbitrary embedding Π'_l and set the list L'' to the empty list (see Figure 3.10).

   (c) p_l **is a** P**-node:** We choose an embedding for G'_l where the vertices r_{l-1} and r_l are on the same face. Such an embedding is easy to find, because the two vertices of the split pair that defines p_l are both adjacent to r_{l-1} and r_l (see Figure 3.11). We define L'' as the empty list of edges.

(d) p_l **is an** R**-node:** We choose the embedding Π'_l for G'_l that has been used in the algorithm to define the list L'. In this case, our list L'' and the list L' computed by the algorithm are identical (see Figure 3.12).

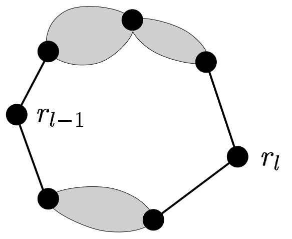


Figure 3.10: Example for the structure of the graph G'_l in the case where p_l is an S-node

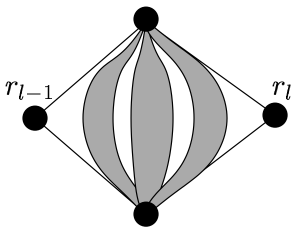


Figure 3.11: Example for the structure of the graph G'_l in the case where p_l is a P-node

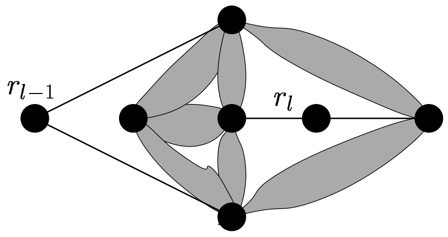


Figure 3.12: Example for the structure of the graph G'_l in the case where p_l is an R-node

Now we have a path L'' that defines an edge insertion path in G'_l from r_{l-1} to r_l with respect to Π'_l. We know by induction that L_{l-1} is a prefix of L and because L'' is either empty or was computed by our algorithm, the list L_l defined as L_{l-1} concatenated with L'' is also a prefix of L. This prefix consists of the edges in L that are contained in the graph G_l.

Now we have to construct an embedding Π_l for G_l such that L_l is an edge insertion path for v_1 and r_l with respect to Π_l. It is clear that we get the graph G_l by identifying the vertices r_{l-1} and the two incident edges in G'_l and G_{l-1} and then deleting r_{l-1}. To get the right embedding for the combined graph, we combine the two embeddings Π_{l-1} and Π'_l.

Let x_1 be the last edge in L_{l-1} if L_{l-1} is non empty and v_1 otherwise. Let x_2 be the first edge in L'' if it is non empty and r_l otherwise. Then there is a face F_1 in Π_{l-1} with x_1 and r_{l-1} on its boundary. Additionally, there is a face F_2 in Π'_l with x_2 and r_{l-1} on its boundary. This follows from the fact that L_{l-1} is an edge insertion path in G_{l-1} for v_1 and r_{l-1} with respect to Π_{l-1} and the fact that L'' is an edge insertion path in G'_l for r_{l-1} and r_l with respect to Π'_l.

We now want to construct an embedding Π_l for G_l such that there is a face in Π_l with x_1 and x_2 on its boundary. Note that boundaries of faces are directed cycles and therefore define a direction for each edge on the boundary (we assume that a face boundary is always directed such that the face is to the right of the boundary). We consider two different cases, according to the directions in which the face boundaries of F_1 and F_2 traverse the edges incident to r_{l-1}.

(a) **The two directions are different.** This situation is shown in Figure 3.13. In this case, we can simply merge the embeddings Π_{l-1} and Π'_l by identifying r_{l-1} together with the two adjacent vertices and then deleting r_{l-1}. The corresponding embedding Π_l contains a face F with x_1 and x_2 on its boundary that was constructed by merging F_1 and F_2.

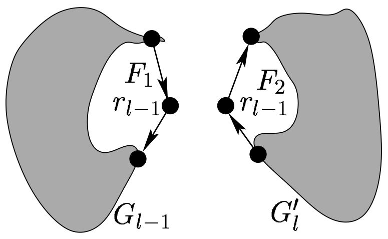


Figure 3.13: The case where r_{l-1} is passed in different directions by the boundaries of F_1 and F_2

(b) **The two directions are the same.** This situation is shown in Figure 3.14. In this case, we construct the mirror embedding of Π_{l-1} by reversing all face cycles. This implies reversing for all vertices the list of incident edges in clockwise order. The list L_{l-1} is also an edge insertion path for v_1 and r_{l-1} in the new embedding Π'_{l-1}, because we just mirrored the embedding. But the boundary of face F_1 traverses the edges incident to r_{l-1} in the opposite direction as F_2 and we can use the same merging operation as in the first case.

We now look at the list L_l that we get by concatenating L_{l-1} and L''. This list is an edge insertion path for v_1 and r_l with respect to Π_l. The reason is that L_{l-1} is an edge insertion path for v_1 and r_{l-1} in G_{l-1} while L'' is an edge insertion path for r_{l-1} and r_l in G'_l. By mirroring the embedding Π_{l-1} of G_{l-1} if necessary, we make sure that the faces F_1 and F_2 get merged to a single face F with the property that x_1 (either the last edge in L_{l-1} or v_1) and x_2 (either the first edge in L'' or r_l) are both on the boundary of F.

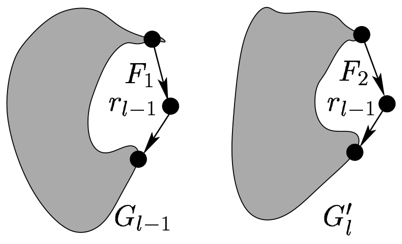


Figure 3.14: The case where r_{l-1} is passed in the same direction by the boundary of F_1 and F_2

Thus, the concatenation of L_{l-1} and L'', which we now call L_l, is an edge insertion path for v_1 and r_l with respect to Π_l (see Figure 3.15 for an illustration).

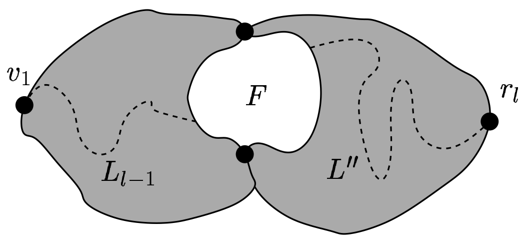


Figure 3.15: The completed merger of Π_{l-1} and Π_l.

We can now show that L is an edge insertion path for v_1 and v_2 with respect to some embedding Π of G. We know that a prefix L_{k-1} of L ($L = L_k$) is an edge insertion path for v_1 and r_{k-1} in G_{k-1} with respect to embedding Π_{k-1}. We define G'_k as the graph that we get by expanding every edge in the skeleton S of p_k except the representative $e_1 = (u_1, u_2)$ of v_1 and then replacing e_1 by a path of length two with vertex r_{k-1} as the middle vertex. With the same argument that we used when dealing with node p_1 we can exclude the case that p_k is a Q- or P-node. So there are only two cases left.

(a) **p_k is an S-node:** In this case, our algorithm does not look at node p_k. It will not add any edges to L that are not contained in L_{k-1}. Since we know that L_{k-1} is the prefix of L containing the edges in L that are also contained in G_{k-1}, we can conclude that L and L_{k-1} are in fact identical. We will show that every embedding Π of G produced through an arbitrary extension of Π_{k-1} has the property that L is an edge insertion path for v_1 and v_2 with respect to Π.

The graph G'_k shares only the vertex r_{k-1} and its two neighbors with G_{k-1}. The skeleton S of p_k has only one embedding with the two faces F_A and F_B. The vertices v_2, r_{k-1} and its two neighbors are on the boundary of both faces. The two edges incident to r_{k-1} are also on the boundary of both faces. When we expand the edges of S (the skeleton of p_k) and construct an arbitrary embedding Π'_k for G'_k, the faces F_A and F_B may get more edges and vertices on their boundary, but the four vertices (v_2, r_{k-1} and its two neighbors) will stay on the boundary of the faces and the two edges incident to r_{k-1} are still part of the boundary of the faces (see Figure 3.16).

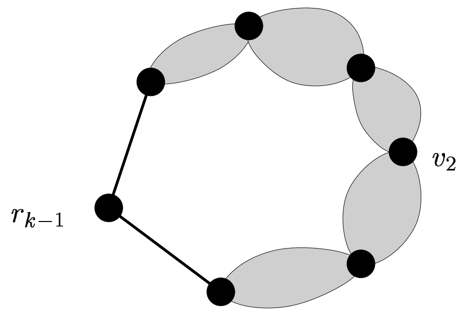


Figure 3.16: Example for the structure of the graph G'_k for the case that the last node on P_2 is an S-node.

Again we define x_1 as the last edge e in L_{k-1} if it exists or as v_1 otherwise. There is a face F in Π_{k-1} with x_1 and r_{k-1} on its boundary. So when we merge Π_{k-1} with Π'_k to produce the embedding Π of G, F will either be merged with F_A or F_B. In both cases, x_1 will be on the boundary of the same face as v_2 and we can be sure that L is in fact an edge insertion path for v_1 and v_2 with respect to Π.

(b) p_k **is an** R**-node:** In this case, our algorithm will construct the graph G'_k by expanding every edge in the skeleton S of p_k except for the representative e_1 of v_1 and splitting e_1 by inserting vertex y_1 (which we call r_{k-1} in this proof). The algorithm will then compute an edge insertion path L' for r_{k-1} and v_2 with respect to some embedding Π'_k of G'_k.

Since L_{k-1} is the prefix of L that contains the edges in L that are contained in G_{k-1}, we know that the rest of L (called $\overline{L_{k-1}}$) is L'. We can use the same argumentation as before to show that the merging of Π'_k with either Π_{k-1} or its mirror embedding will result in an embedding Π with the property that the list L that we define as the concatenation of L_{k-1} and L' is an edge insertion path for v_1 and v_2 with respect to Π.

□

When we have an edge insertion path L for the two vertices v_1 and v_2, there is a simple way for finding an embedding Π such that L is an edge insertion path for v_1 and v_2 with respect to Π. We just split every edge e_i in L by introducing a new vertex s_i. Then we connect the vertices v_1, v_2 and s_i with new edges to form a path p that starts at v_1, passes the vertices s_1 to s_k and ends at vertex v_2.

Since L is an edge insertion path, the graph G' generated by this operation is planar. Therefore, we can compute a combinatorial embedding Π' for G'. Algorithms for computing embeddings in linear time have been developed by Hopcroft and Tarjan (31) (the embedding phase of the algorithm is also treated in (41) and (39) with improvements by Booth and Lueker (13). When we delete the edges on p from Π' and replace all the vertices s_i and their incident edges with the original edges, we get a combinatorial embedding Π for G with the property that L is an edge insertion path for v_1 and v_2 with respect to Π.

What we have shown in the proof above is that the list of edges computed by Algorithm 1 is indeed an edge insertion path for v_1 and v_2 with respect to some embedding Π of G. But we still have to show that this edge insertion path has minimum length among all edge insertion paths with respect to any embedding of G. Therefore, we need the following theorem.

Theorem 3.3 *The edge insertion path computed by Algorithm 1 is optimal.*

Proof We have already seen that Algorithm 1 computes an edge insertion path for v_1 and v_2, so let L be the computed path with respect to embedding Π. We have to show that each edge insertion path for v_1 and v_2 has length at least $|L|$. Let L' be an optimal edge insertion path and let Π' be an embedding such that L' is an edge insertion path in Π'. We show that $|L'| \geq |L|$ holds.

Consider one step of the while-loop in Algorithm 1. Let μ be the R-node popped from P_3 and S its skeleton. The algorithm marks the edges in S that are representatives of v_1 or v_2. Denote with G_μ the graph obtained from S by expanding all unmarked edges in S. The embedding Π' implies an embedding Π'_μ of G_μ. Using this embedding, we can construct a dual graph $\overline{S_{\Pi'}}$ of G_μ augmented with x_1 and x_2.

First we show that the edges of L' that are contained in G_μ form a subsequence of L'. If both vertices v_1 and v_2 are contained in S, then all edges of L' are edges of G_μ. If L' does not contain edges of G_μ, our claim is also true. Now we assume that L' contains edges of G_μ. If v_1 is not contained in S then it is represented by edge $e_1 = (s_1, t_1)$. The expansion graph $G(e_1)$ of e_1 contains v_1. The same is true for v_2. If it is not contained in S, it is represented by edge $e_2 = (s_2, t_2)$ and the expansion graph $G(e_2)$ contains v_2. There are two faces F_1 and F_2 in Π' that contain edges of $G(e_1)$ and of G_μ on their boundary. This is true because $G(e_1)$ is connected to G_μ via the separation pair $\{s_1, t_1\}$. This situation is depicted in Figure 3.17. This Figure shows the graph G where G_μ is colored light grey and the two graphs $G(e_1)$ and $G(e_2)$ are colored dark grey.

Lets assume that the edges of G_μ contained in L' do not form a subsequence of L'. There are several different situations with this property. Here, we will only treat one of the cases, the other cases can be treated similarly. We consider the situation shown in Figure 3.17. L' is represented by the dashed line. This line leaves $G(e_1)$, travels through G_μ, reenters $G(e_1)$ and then leaves it for good. So L' looks as follows:

$$L' = L_1 \cdot L_2 \cdot L_3 \cdot L_4 \cdot L_5$$

The sub-path L_1 is the part of L' starting at v_1 and ending where L' leaves $G(e_1)$ for the first time. The part between the first entry of L' into G_μ and the re-entry into $G(e_1)$ is L_2. The sub-path L_3 is the last sub-path of L' that contains edges of $G(e_1)$. The last sub-path of L' that contains only edges of G_μ is L_4. The last part of L' that contains only edges of $G(e_2)$ and ends in v_2 is L_5.

The lists L_1 and L_3 contain only edges of $G(e_1)$ while the lists L_2 and L_4 contain only edges of G_μ. The list L_5 contains only edges of $G(e_2)$. We claim that $L_1 \cdot L_4 \cdot L_5$ is also an edge insertion path for v_1 and v_2 in G. This is true because we can rotate $G(e_1)$ around the axis given by the vertices s_1 and t_1 to produce an embedding of G where the last edge of L_1 is on

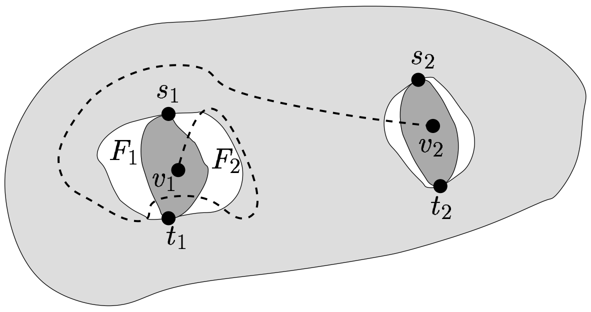


Figure 3.17: An edge insertion path where the edges of G_μ do not form a subsequence cannot be an optimal edge insertion path.

the boundary of the same face as the first edge of L_4. Figure 3.18 shows G after the rotation of $G(e_1)$.

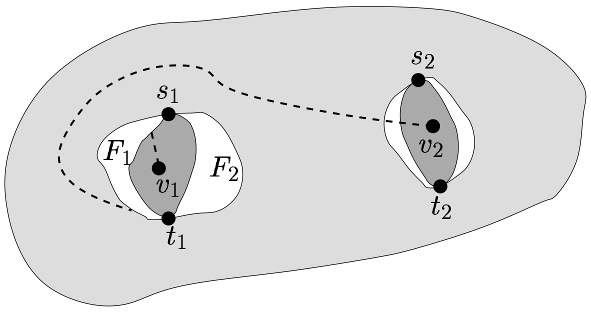


Figure 3.18: The situation of Figure 3.17 after rotating $G(e_1)$.

We can always shorten an edge insertion path that reenters $G(e_1)$ because there are exactly two faces F_1 and F_2 whose boundaries contain edges of G_μ *and* edges of $G(e_1)$. So every edge insertion path that starts in $G(e_1)$ and passes through G_μ must either pass through F_1 or F_2. By rotating $G(e_1)$, we can always make sure that we still have an edge insertion path when we remove a suitable sequence of edges from the original edge insertion path.

When applied to the edge insertion path L', we have a contradiction to our assumption that L' is an optimal edge insertion path for v_1 and v_2, because we can find a shorter edge insertion path. It follows that the edges on L' that belong to G_μ must form a subsequence L'_μ of L'.

L'_μ has the property that the corresponding dual edges plus two additional edges (one starting at x_1, and one ending at x_2) form a path $\widetilde{p}$ in $\overline{S_{\Pi'}}$ from x_1 to x_2. We can assume that $\widetilde{p}$ is indeed a path and thus contains no cycle, since the path obtained from deleting all edges in a cycle would imply an even shorter edge insertion path for v_1 and v_2.

Consider now an unmarked edge $e \in S$ with the property that $\widetilde{p}$ contains at least one edge of the expansion graph $G(e)$ of e. There are exactly two faces, say $F_1(e)$ and $F_2(e)$, in Π'_μ

where the boundary contains edges of G_μ as well as edges from the rest of $G(e)$. The dual counterparts of the edges in $L' \cap G(e)$ must form a sub-path $\widetilde{p(e)}$ of $\widetilde{p}$ leading from $F_1(e)$ to $F_2(e)$ (or vice versa). This follows from the optimality of L' and the fact that $\widetilde{p}$ contains no cycle.

If we denote with $T(e)$ the traversing costs of e, then $\widetilde{p(e)}$ must have length $T(e)$. Because of this fact, we can simplify the rest of this proof by looking only at the dual graph of the skeleton S of μ where the weight of each edge is equal to the traversing costs of the edge. So let $\bar{S}'$ be the dual graph of S augmented by x_1 and x_2 where the weight of each unmarked edge is given by the traversing costs of the corresponding edge in S. Note that we do not have to specify a combinatorial embedding for which the dual graph was created since the two embeddings of the triconnected graph S are mirror images of each other and the corresponding dual graphs are isomorphic to each other.

We transform $\widetilde{p}$ into the corresponding path p' in $\bar{S}'$. This is done by replacing each $\widetilde{p(e)}$ for each unmarked edge e with $\bar{e}$ (the dual edge of e) and adding two edges that have been introduced when we augmented $\bar{S}$ with x_1 and x_2. Now we have a path p' in the weighted graph $\bar{S}'$ connecting x_1 and x_2. Since the weights of the edges added during the augmentation are zero, the total costs of p' in $\bar{S}'$ are $|L'_\mu|$.

On the other hand, the list L_μ of edges computed by Algorithm 1 (which is called L' in the algorithm and is appended to L at the end of the body of the while loop) implies a shortest path p with the same length in the weighted graph $\bar{S}$, which is obtained analogously to $\bar{S}'$. This is true, because the traversing costs of an edge are independent of the embedding of the respective expansion graph by Theorem 3.1. Note that $\bar{S}$ and $\bar{S}'$ are either identical or mirror images of each other. Therefore, the length of p' is at least the length of p, and $|L'_\mu| \geq |L_\mu|$ holds.

The length of L is the sum of the length of all L_μ. Since all G_μ are edge disjoint, it follows that the length of L' is at least $\sum_{\mu \in P} |L'_\mu|$, and therefore $|L'| \geq |L|$ must hold. □

3.3.3 Running time

We show that the running time of our algorithm is linear in the size of the graph. First, we clarify what is meant by this statement. A planar graph has at most $3n - 6$ edges where n is the number of vertices (if we assume that there are no multi-edges and no self loops). So there can be no misunderstanding when we say that the running time of the algorithm is linear in the size of the graph.

We look at Algorithm 1 line by line and show that each action can be performed in linear time (linear in the size of the input graph). Computing the SPQR-tree of a biconnected planar graph can be done in linear time (6; 27). Finding allocation nodes μ_1 and μ_2 for the vertices v_1 and v_2 can also be done in linear time by looking at all skeletons of all nodes of the SPQR-tree. Note that the size of an SPQR-tree for a planar graph (including the skeletons) is linear. Finding the path P_1 from μ_1 to μ_2 in the tree can be done in linear time using depth first search.

We transform P_1 into P_2 by deleting nodes from both ends of the path until we have produced the shortest path from an allocation node of v_1 to an allocation node of v_2. This can also be done in linear time. One scan through P_2 is necessary to delete all S- and P-nodes of P_2 and for marking the representatives of the vertices v_1 and v_2 in the skeletons. This also takes linear time.

To analyze the running time of the **while**-loop, it is important to note that all graphs computed in the loop are disjoint except that they may share two vertices if the two R-nodes in question are connected by an edge in the SPQR-tree. Constructing the union of all the graphs will result in a subgraph of the original graph G (if we delete the artificial edges and nodes generated by the splitting of edges performed in the algorithm). So the sum of the sizes of all graphs considered in the algorithm is at most the size of the original graph plus at most a linear number of vertices and edges introduced by the algorithm.

Now we look at a single iteration of the loop. If an edge splitting operation is necessary, it can be done in constant time. Then we replace all unmarked edges by their expansion graphs. If the current R-node treated in the loop is r, then we call the graph generated by the expansion of the unmarked edges G_r. Computing an arbitrary embedding for G_r can be done in linear time, using for example the algorithm described in (41). The same holds for the computation of the dual graph and for augmenting the dual graph. Computing a shortest path in the extended dual graph of G_r can also be done in linear time using breadth first search. We do not have to set the weights of the edges added in the augmentation step to zero, because each path from x_1 to x_2 has to use exactly two of these edges. Since all edges have weight one, we can use breadth first search to find the shortest path.

Deleting the head and tail of the path to get the edge insertion path can be done in constant time while the translation from dual edges to primal edges takes linear time in the size of G_r. Appending the resulting path to the result list is possible in constant time. Therefore the running time of Algorithm 1 is linear in the size of the biconnected planar graph G. This is also true in practice, as our computational results in Section 3.5 on page 60 show.

3.4 Generalization to Arbitrary Graphs

If we have an algorithm that works for connected graphs, it is very easy to extend the algorithm to arbitrary graphs. If the two vertices that we want to connect are contained in the same connected component, we can apply the algorithm used for connected graphs. If they do not belong to the same connected component, we can always insert an edge between them without losing planarity.

This can be seen as follows: Let G_1 be the connected component containing vertex v_1 and G_2 the component containing vertex v_2. We can always construct a drawing of G_1 with v_1 on the outer face and a drawing of G_2 with v_2 on the outer face. Then we can connect v_1 and v_2 with an edge through the global outer face of the graph.

So the missing link is the algorithm for connected graphs. This algorithm, a proof of correctness and the running time are given in the rest of this section.

3.4.1 The Algorithm For Connected Graphs

First we consider the case where G has more than one *block* (a block is a maximal biconnected subgraph), but both vertices we want to connect are contained in the same block b. We claim that all optimal edge insertion paths in G contain only edges of b.

Theorem 3.4 *Let G be a connected planar graph and p an edge insertion path for (v_1, v_2) with respect to some embedding Π of G. If both v_1 and v_2 belong to the same block b of G and if p includes an edge contained in a different block b', than p is not an optimal edge insertion path.*

Proof Let $s = b_1, b_2, \ldots, b_k$ be a sequence of all the blocks of G with the following properties:

1. $b_1 = b$
2. The graph G_i that we get as the union of the blocks b_1 to b_i is connected for all $1 \leq i \leq k$.

Such a sequence must exist since G is connected. We define the embedding Π_i of G_i as the embedding that we get by deleting all blocks b_j with $j > i$ from Π. So Π_k and Π are identical. We will now prove the following fact by induction over i: An optimal edge insertion path connecting v_1 and v_2 in G_i contains only edges in b.

- $i = 1$: The claim follows immediately since b is the only block in G_1.
- $i > 1$: We know that Π_{i-1} differs from Π_i only in the fact that b_i is missing. Since G_i is planar, adding b_i to Π_{i-1} can only be done by inserting all the vertices and edges of b_i into a face F of Π_{i-1}. This is true because b_i is connected and inserting two vertices of b_i into different faces of Π_{i-1} must produce a crossing.

  Let Π_{b_i} be the embedding of b_i that we get by deleting all vertices and edges of Π that do not belong to b_i. Since G_{i-1} is connected and Π_i is a combinatorial embedding, it follows with the same argumentation as above that in Π_i, Π_{i-1} is embedded into a single face F' of Π_{b_i}.

  From the construction of Π_i we know that its faces can be partitioned into three sets:

  1. The set F_1 of faces of Π_{i-1} without F.
  2. The set F_2 of faces of Π_{b_i} without F'.
  3. The face $\hat{F}$ we get by merging F and F'.

  By merging F and F' we mean the following process: Let $c, u_1, u_2, \ldots, u_k$ be the sequence of vertices on the boundary of F such that F is on the right of the directed cycle of edges defined by this sequence. Vertex c is the cut-vertex separating G_{i-1} from b_i. Let $c, w_1, w_2, \ldots, w_l$ be the sequence of vertices on the boundary of face F' such that F' is on the right of the directed cycle of edges defined by this sequence. Then the sequence of the vertices on the boundary of $\hat{F}$ is $c, w_1, w_2, \ldots, w_l, c, u_1, u_2, \ldots, u_k$ (again $\hat{F}$ is to the right of the cycle defined by the sequence). See Figure 3.19 for an example.

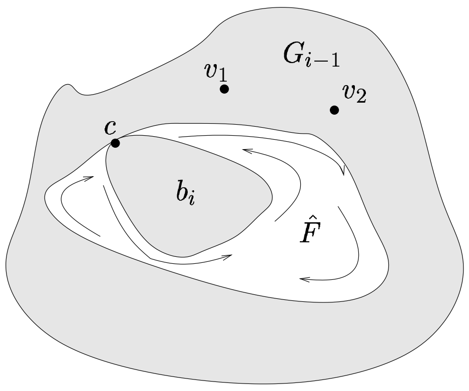


Figure 3.19: Adding Π_{b_i} to the embedding Π_{i-1} to get embedding Π_i

Now we look at the dual graph D_i of Π_i augmented by v_1 and v_2. How can we construct D_i from the augmented dual graph D_{i-1} of Π_{i-1} and the dual graph D_{b_i} of Π_{b_i}? Since we have one vertex in D_i for every face and for v_1 and v_2, the vertices of D_i are v_1 and v_2 together with one vertex corresponding to each element in the sets F_1 and F_2 and the vertex representing $\hat{F}$.

For simplicity, we define V_1 as the set of vertices in D_i representing faces in F_1 together with the vertices v_1 and v_2, V_2 as the set of vertices in D_i representing faces in F_2 and $\hat{v}$ as the vertex representing $\hat{F}$.

By construction of Π_i it is not hard to see that we construct D_i from D_{i-1} and D_{b_i} by merging the vertex corresponding to F in D_{i-1} and the vertex corresponding to F' in D_{b_i} to produce the new vertex $\hat{v}$. Thus the following statements are true:

1. There is no edge in D_i from a vertex in V_1 to a vertex in V_2.
2. All paths from a vertex in V_1 to a vertex in V_2 must pass $\hat{v}$.
3. Every path in D_{i-1} connecting v_1 and v_2 which does not include the vertex representing face F is also present in D_i.
4. If a path p connecting v_1 and v_2 in D_{i-1} includes the vertex corresponding to face F, we can construct a path of the same length connecting v_1 and v_2 in D_i by replacing the vertex representing F in p by the vertex $\hat{v}$.

The last two facts guarantee that for every shortest path p in the augmented dual graph D_{i-1} connecting v_1 and v_2, we can find a path of equal length in D_i which does not use any vertex in V_2. The first two facts guarantee that there is no edge insertion path for v_1 and v_2 that uses vertices in V_2 because the only way of getting from a vertex in V_1 to another vertex in V_1 via a vertex in V_2 is by passing $\hat{v}$ twice. Thus we have proven our claim.

□

This proves that if we want to connect two vertices in the same block b of the graph, it suffices to apply our algorithm for biconnected graphs to the block b.

Now we have to look at the case where v_1 and v_2 are in different blocks of the graph. In the following algorithm, we use the function `OptimalBlockInserter` (Algorithm 1 on page 42) as a subroutine. An important data structure used in the algorithm is the *block tree* of a graph. This data structure was already used by Westbrook and Tarjan (49) and Tamassia (48). (5) extended the tree by associating the SPQR-tree of each block with the corresponding nodes in the tree. This tree has two types of nodes: b-nodes and c-nodes. There is one b-node for every block in the graph and one c-node for every vertex. There is an edge connecting a c-node and a b-node if and only if the block represented by the b-node contains the vertex represented by the c-node.

Figure 3.20 shows a graph and its block tree. The graph in Figure 3.20(a) has five blocks that are drawn as grey squares in the drawing of the block tree (Figure 3.20(b)). The cut vertices in the graph are the c-vertices in the block tree with degree greater one.

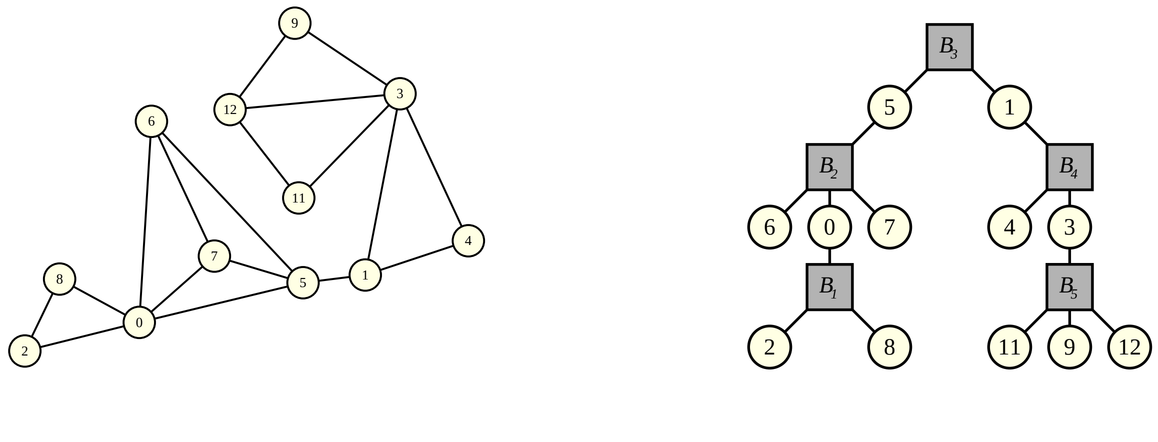


Figure 3.20: A graph and its block tree. The b-vertices of the block tree are drawn as grey squares.

Another important term the algorithm uses is the *representative* of vertex v in a block B. We look at the shortest path P from the b-node corresponding to B in the block tree to the c-node corresponding to v. Let c be the first c-node on this path (as seen from the b-node corresponding to B). Then the vertex corresponding to c is the representative of v in B. Note that this vertex is v itself if v is contained in the block B. Our algorithm for treating connected graphs works as follows:

Algorithm 2: Algorithm for edge insertion in connected planar graphs

```
Input: Planar connected graph $G = (V, E)$ and two non-adjacent vertices $v_1$ and $v_2$ in
       $G$.
Result: An edge insertion path $L$ for $v_1$ and $v_2$.
begin
    Build the block tree $\mathcal{B}$ of $G$;
    Compute the path $P$ form $v_1$ to $v_2$ in $\mathcal{B}$;
    Set $B = (b_1, b_2, \dots, b_k)$ to the list of blocks on $P$;
    Set $L$ to an empty list of edges;
    for $i \leftarrow 1$ to $k$ do
        Let $x_1$ and $x_2$ be the representatives of $v_1$ and $v_2$ in $B_i$;
        $L'$ = OptimalBlockInserter($B_i, x_1, x_2$);
        Append $L'$ to $L$;
end
```

3.4.2 Correctness and Optimality

The proof of correctness and optimality of Algorithm 2 relies on the correctness and optimality of Algorithm 1.

Theorem 3.5 *Let G be a planar connected graph and let v_1 and v_2 be two non adjacent vertices in G. Then the list L of edges computed by Algorithm 2 is an optimal edge insertion path for v_1 and v_2.*

Proof We use the same approach as in the correctness proof for the algorithm that works on biconnected graphs (Theorem 3.2). We define G_i as the graph that consists of the blocks b_1 to b_i in Algorithm 2 and L_i as the list L after iteration i of the **for**-loop in the algorithm. We will show by induction that L_i is an optimal edge insertion path in G_i for v_1 and the representative y_i of v_2 (in G_k, the representative of v_2 is v_2 itself).

We use induction over i. For each i, we must show that there is an embedding Π_i such that L_i is an edge insertion path for v_1 and y_i with respect to Π_i and that the path has minimum length among all edge insertion path with respect to any embedding of G_i.

1. $i = 1$**:** The correctness follows directly from the correctness and optimality of Algorithm 1. So we know that L_1 is indeed an optimal edge insertion path for v_1 and y_1.

2. $i > 1$**:** We know that the theorem is correct for $i < l$ and we want to prove correctness for $i = l$. Since Algorithm 1 is correct, we know that the list L' computed in iteration l of the algorithm is an optimal edge insertion path in b_l for the representative x_l of v_1 (called x_1 in the algorithm) and the representative y_l of v_2 (called x_2 in the algorithm).

   Let Π'_l be an embedding of b_l such that L' is an edge insertion path with respect to Π'_l. The construction of the block tree guarantees that the vertex x_l is a cut-vertex of G and that x_l and vertex y_{l-1} in G_{l-1} are identical. So the representative of v_2 in G_{l-1} is the

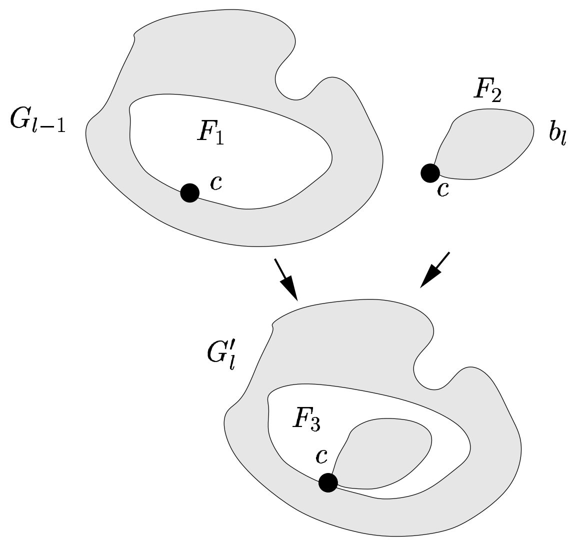


Figure 3.21: Merging the embeddings Π_{l-1} and Π'_l

same vertex as the representative of v_1 in b_l. Since both y_{l-1} and x_l stand for the same vertex, we will call this vertex c in the rest of this proof.

By induction we know that L_{l-1} is an optimal edge insertion path in G_{l-1} for v_1 and c with respect to Π_{l-1}. The way we construct G_l from G_{l-1} and b_l is by identifying the vertex c in both graphs. There is no other vertex or edge that both graphs share since b_l is a maximal biconnected subgraph of G.

How can we construct an embedding Π_l from Π_{l-1} and Π'_l such that L_{l-1} concatenated with L' is an edge insertion path with respect to Π_l? We define α as the last edge in L_{l-1} if the list is not empty and as v_1 otherwise. We define β as the first edge in L' and as y_l if L' is empty. Then there must be a face F_1 in Π_{l-1} with c and α on its boundary. Additionally, there must be a face F_2 in Π'_l with c and β on its boundary. We embed Π'_l into face F_1 of Π_{l-1} such that Π_{l-1} is embedded into face F_2 of Π'_l. This is done just like in the proof of Theorem 3.4 (see also Figure 3.21 on page 58).

So we get embedding Π_l by merging the faces F_1 and F_2 of Π_{l-1} and Π'_l. Let F_3 be the face resulting from the merger of F_1 and F_2. All other faces remain untouched. We do not have to perform a mirror operation as in the proof of Theorem 3.2 because we do not merge two faces that share a virtual edge. In our case, the faces only share a single vertex.

We know that L_{l-1} is an edge insertion path in G_l for v_1 and c with respect to Π_l and that L' is an edge insertion path in G_l for c and y_l. The merging operation guarantees that α and β are both on the boundary of face F_3 in Π_l and that L_{l-1} and L' are still edge insertion paths for the same pairs of vertices. We can get from α to β in Π_l without crossing another edge. Thus, L_l which is defined as the concatenation of L_{l-1} and L_l, is an edge insertion path in G_l for v_1 and y_l with respect to Π_l.

Now we assume that there is a shorter edge insertion path $\hat{L}$ for v_1 and y_l with respect to some embedding $\hat{\Pi}$ of G_l. There is exactly one face $\hat{F}$ in $\hat{\Pi}$ where the boundary includes edges from G_{l-1} as well as edges from b_l. The sequence of the vertices on the boundary of this face must include the vertex c two times because c is a cut-vertex.

Let $\bar{G}_l$ bet the dual graph of G_l with respect to embedding $\hat{\Pi}$. The edge insertion path $\hat{L}$ defines a path $\bar{L}$ in $\bar{G}_l$. $\bar{L}$ passes the vertex of $\bar{G}_l$ corresponding to $\hat{F}$ exactly once. The path $\bar{L}$ must pass it at least once because this vertex is a cut-vertex in $\bar{G}_l$ separating the part of G_l that contains v_1 and the part that contains v_l. It cannot pass it several times, because it is a path.

Therefore we can split $\hat{L}$ into the two parts $\hat{L}_A$ and $\hat{L}_B$ where the first part contains only edges from G_{l-1} and the second part contains only edges from b_l. We define Π_A as the embedding for G_{l-1} that we get by deleting all nodes and edges from $\hat{\Pi}$ that are not contained in G_{l-1} and Π_B as the embedding for b_l that we get by deleting all nodes and edges from $\hat{\Pi}$ that are not contained in b_l. In other words, Π_A is the embedding of G_{l-1} defined by $\hat{\Pi}$ and Π_B the embedding of b defined by $\hat{\Pi}$. Then $\hat{L}_A$ is an edge insertion path in G_{l-1} for v_1 and c with respect to Π_A and $\hat{L}_B$ is an edge insertion path in b_l for c and y_l with respect to Π_B.

Since we assume that $\hat{L}$ is shorter than L_l, at least one of the following statements must be true:

(a) $\hat{L}_A$ is shorter than L_{l-1}.

(b) $\hat{L}_B$ is shorter than L'.

The first statement is a contradictions against the fact that L_{l-1} is an optimal edge insertion path for v_1 and c in G_{l-1}. The second statement is a contradiction to the correctness and optimality of Algorithm 1 because we used it to compute the shortest edge insertion path in b_l connecting c and y_l. So $\hat{L}$ cannot be shorter than L_l and we know that L_l is an optimal edge insertion path for v_1 and y_l in G_l.

This concludes the induction where we have shown that L is an optimal edge insertion path in G_k for v_1 and v_2 with respect to some embedding Π_k of G_k. But G_k and G may not be identical. G_k contains only the biconnected blocks of G that belong to the path in the block tree that starts at v_1 and ends in v_2. We have to show that L is also an optimal edge insertion path in G. To see this, we assume that we add the rest of the blocks one by one to G_k such that we always have a connected graph. Lets assume that the sequence of the graphs created in this way is $G_k, G_{k+1}, \ldots, G_m$ where G_m is G.

We can show with the same argument used in the proof of Theorem 3.4 on page 54 that an optimal edge insertion path for v_1 and v_2 in G_i with $k \leq i < m$ is also an optimal edge insertion path for v_1 and v_2 in G_{i+1}. When we add the next block b to some G_i with $i \geq k$, we always do this by merging a face boundary of G_i with a face boundary in b and leaving all other faces untouched. So any two faces or vertices of G_i that were on the boundary of the same face before the merger will also be on the boundary of the same face after the merger. Thus, an edge insertion path for G_i is also an edge insertion path for G_{i+1}.

Adding blocks to a planar graph G_k cannot create a shorter edge insertion path for v_1 and v_k. This can be seen as follows: when we add a block to the graph G_i with $i \geq k$, we also add a block to the dual graph. Let D_1 be the part of the dual graph of G_{i+1} corresponding to G_i and D_2 the part corresponding to b_{i+1}. D_1 is connected to D_2 only via one cut-vertex. Therefore, any path leaving D_1 in the dual graph must do so using this cut-vertex. The path can never reenter D_1 because the only way back is again via the cut-vertex and this is not possible because a path can use a vertex at most once.

Hence, a path in the dual graph of G that connects two vertices in G_k uses only the duals of edges in G_k. So a shortest path in the dual of G_i will also be a shortest path in the dual of G_{i+1}. And an optimal edge insertion path in G between two vertices in G_k will not contain any edges of a block that does not belong to G_k. Thus, L is an optimal edge insertion path in G for v_1 and v_2. $\square$

3.4.3 Running time

Again, we look at the algorithm step by step. Computing the block tree of a graph can be done in linear time using an algorithm for finding biconnected components (such an algorithm can for example be found in (15)). Since the tree has linear size, we can find the path P in linear time using depth first search. In the **for**-loop, we execute Algorithm 1 for every block on P. Each execution needs time linear in the size of the block (see Section 3.3.3 on page 52) and since the blocks are edge-disjoint the whole algorithm needs only linear time.

3.5 Computational Results

We have implemented our algorithm using AGD (AGD), a library of algorithms for graph drawing, that contains a state-of-the-art implementation of the planarization method and a linear time implementation of SPQR-trees (27).

First we compared the performance of the following two algorithms that both insert an edge e into a planar graph G:

- **SPI**: This algorithm is a heuristic. It chooses an arbitrary embedding Π of G and then inserts e by computing a shortest path in the dual graph with respect to Π. The abbreviation **SPI** stands for **S**hortest **P**ath **I**nserter.

- **OEI**: This is the new algorithm we presented in this chapter. It inserts e with the minimum possible number of crossings under the restriction that the edges of G do not cross. The abbreviation **OEI** stands for **O**ptimal **E**dge **I**nserter.

To compare the performance of the two algorithms, we used the set of 11529 graphs collected by Battista, Garg, Liotta, Tamassia, Tassinari, and Vargiu (2). They either originate from industry or are derived from such graphs. There are 8249 non-planar graphs in the benchmark set. Figure 3.22 shows how many graphs the set of non-planar graphs contains for each number of vertices. Figure 3.23 shows the average number of edges in the non-planar graphs with the same number of edges. It seems that the number of edges grows linearly with the number of vertices, so the graphs are quite sparse on average.

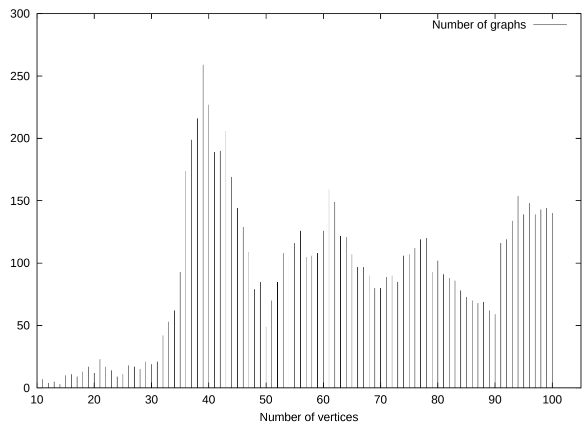


Figure 3.22: Number of non-planar graphs per number of vertices.

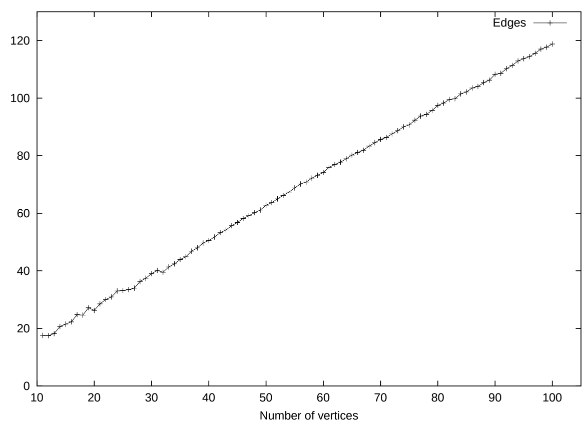


Figure 3.23: Average number of edges for non-planar graphs with the same number of vertices.

For our first test, we generated test instances out of the non-planar graphs by executing the following steps on each of them:

1. We computed a planar subgraph G' of the graph G using the class `SubgraphPlanarizer` of the AGD-library (see **(author?)** (AGD)). It finds a planar subgraph using PQ-trees as described by Jünger, Leipert, and Mutzel (35). Note that the subgraph computed is not necessarily a maximal planar subgraph.
2. Let L be the set of edges that are contained in G but not in G'. We insert the edges of L into G' one by one until the resulting graph is not planar anymore. This must happen at

some point because G' is planar while G is not planar. So this step computes a sequence $S = (G_0, G_1, \ldots, G_k)$ of graphs where $G_0 = G'$ and all graphs G_i with $i < k$ are planar subgraphs of G while G_k is a non-planar subgraph.

Let e be the edge of L we inserted to construct G_k from G_{k-1}. Note that G_k has skewness one, which means that the deletion of the single edge e results in a planar graph. The test instance for the two algorithms consists of G_{k-1} and e. So each algorithm is used to insert edge e into the planar graph G_{k-1} thereby producing at least one crossing because G_k is not planar.

Figure 3.24 shows the average size of the set L (the set of edges we delete to make the graph planar) for each number of vertices. This number of deleted edges grows at least linearly with the size of the graphs.

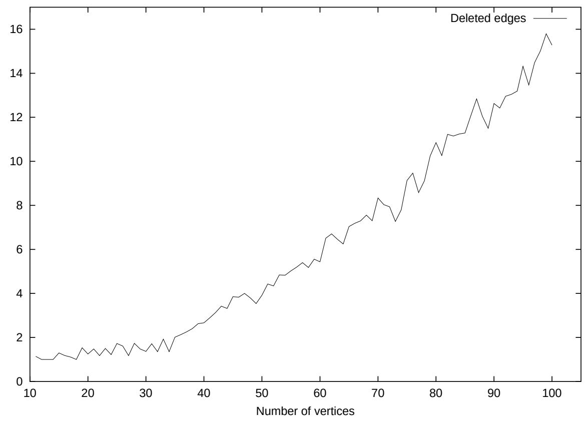


Figure 3.24: Average number of edges deleted to produce a planar subgraph.

Applying the algorithm above to the 8249 non-planar graphs in the benchmark set produced as many problem instances. Each instance consists of a planar graph and an additional edge that makes the graph non-planar. We applied the algorithms **SPI** and **OEI** to each of the instances.

Figure 3.25 shows for each number of vertices the average number of crossings produced for problem instances with that number of vertices by the algorithms **SPI** and **OEI**. The x-axis shows the number of vertices while the y-axis shows for both algorithms the average number of crossings produced using the algorithm on the graphs with the corresponding number of vertices. The figure shows that **OEI** produces significantly less crossings than **SPI**.

Let c_s be the number of crossings produced when applying **SPI** to an instance and c_o the number of crossings produced when applying **OEI**. For each instance, we computed the absolute improvement I_A as $I_A = c_s - c_o$. This is the number of crossings we save if we use **OEI** instead of **SPI**. We define the relative improvement I_R as the following value:

$$I_R = \frac{I_A}{c_s} 100\%$$

The average value of I_A over all instances is 0.28. So we can save 0.28 crossings on average by using **OEI** instead of **SPI**. That does not sound very impressive but the value of I_R is 12.15%. So the average percentage of crossings saved is more than 12 percent if we insert just one edge. The greatest number of crossings saved for an instance (the maximum value for I_A) was three. The maximum percentage saved for an instance (the maximum value for I_R) was 75%. Both maxima occurred for the same problem instance. The graph in question had 66 vertices and 85 edges before the insertion of the additional edge. Inserting the edge with **SPI** produced four crossings, while inserting the edge with **OEI** produced only one crossing.

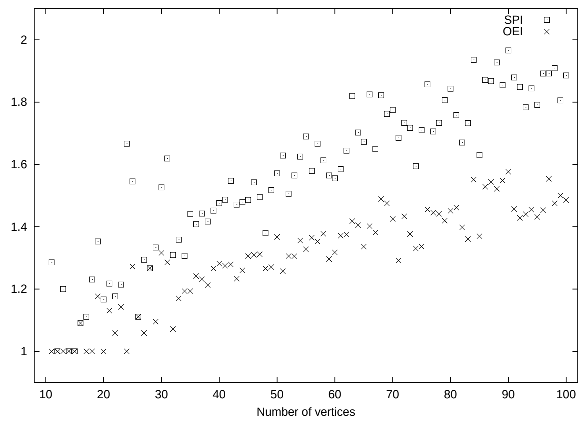


Figure 3.25: Average number of crossings produced by **SPI** and **OEI** over all graphs with the same number of vertices.

We also measured for each instance the time used by both algorithms and computed the average over all instances with the same number of vertices. Additionally, we measured for each graph the time needed to compute an SPQR-tree for each of its non-trivial biconnected components and again computed the average over all graphs with the same number of vertices. The results are shown in Figure 3.26. The figure not only contains the points giving the average time for each number of vertices for the two algorithms **SPI** and **OEI** but also the two straight lines `approx1` and `approx2`. These lines show that the running time of both algorithms is indeed linear. The algorithm **SPI** is much faster because it does not build SPQR-trees for each graph. The figure shows that almost half the computation time of **OEI** is spent for the computation of the SPQR-trees.

We also wanted to know how good the two algorithms **SPI** and **OEI** perform when used in the planarization method. Algorithm 3 gives a short overview of the planarization method.

We programmed two versions of the planarization method that only differ in the method they use for computing the edge insertion path in the **while** loop of Algorithm 3. In the standard algorithm, this is done using **SPI**. In our modified version of the planarization method, we used **OEI** instead of **SPI**. Note that our modified planarization method has the same asymptotic running time as the standard planarization method since **OEI** has linear running time.

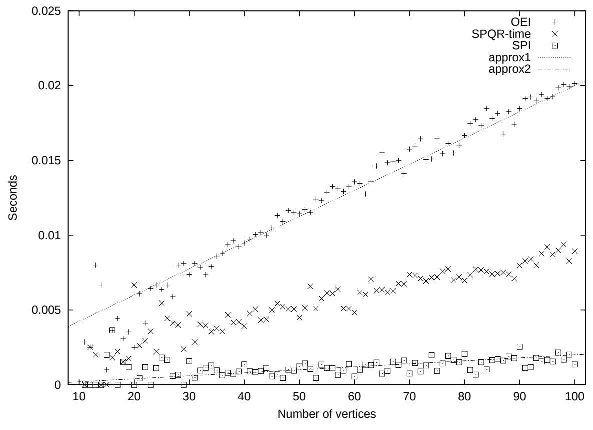


Figure 3.26: Average time used by **SPI** and **OEI** over all graphs with the same number of vertices together with the time needed to compute the SPQR-trees.

Algorithm 3: The planarization method

```
Input: Connected graph $G = (V, E)$
Result: A drawing $D$ of $G$ with a small number of crossings
begin
    Compute a preferably large planar subgraph $G' = (V, E')$ of $G$;
    Set $E_M = E - E'$;
    while $E_M \neq \emptyset$ do
        Let $e = (v_1, v_2)$ be an edge in $E_M$;
        Delete $e$ from $E_M$;
        Find an edge insertion path $p$ for $v_1$ and $v_2$ in $G'$;
        Split the edges on $p$ by inserting artificial vertices;
        Insert a path into $G'$ from $v_1$ to $v_2$ via the artificial vertices;
    Produce a planar drawing $D$ of $G'$;
    Replace all artificial vertices in $D$ by crossings;
end
```

For comparing the performance of the two versions of the planarization method, we use the same 8249 non-planar graphs we used in the experiment where we only inserted one edge. Again, we computed a planar subgraph using the AGD implementation of the algorithm by Jünger, Leipert, and Mutzel (35). Notice that since we insert in general more than one deleted edge in the second phase of the planarization method, the impact of optimally inserting a single edge is not obvious.

Figure 3.27 shows for each number of vertices and the two versions of the planarization method using **SPI** and **OEI** the average number of crossings. The line representing the crossings produced using **SPI** is clearly above the line for **OEI**.

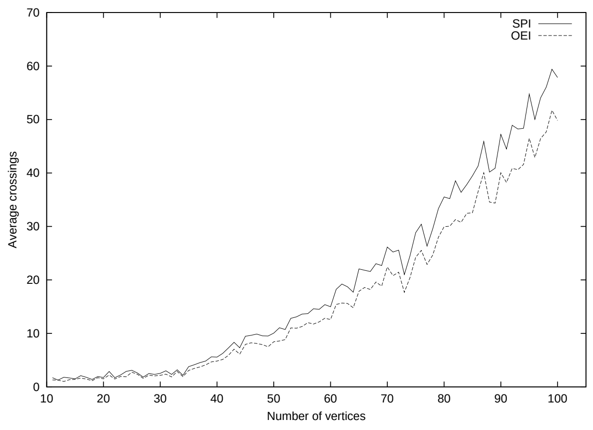


Figure 3.27: Average number of crossings produces by **OEI** and **SPI** for non-planar graph with the same number of vertices.

Figure 3.28 visualizes the significant improvement achieved by our new algorithm. It displays for each number of vertices the average relative improvement I_R of the number of crossings in percent. Let c_s denote the number of crossings produced by the standard algorithm and c_n denote the number of crossings produced by our modified algorithm. Then the value I_R is computed as $I_R = \frac{c_s - c_n}{c_s} 100\%$.

The x-axis in Figure 3.28 shows the number of vertices, while the y-axis shows the average I_R-value over all graphs with the corresponding number of vertices. The average value of I_R varies widely for small graphs because there is only a small number of these graphs in the benchmark set. But for larger graphs, the average I_R value seems to stabilize at about 14%.

For 68% out of the 8249 tested non-planar graphs, the crossing number was smaller using our new method and for only 8% it was greater. This can happen because inserting one edge with the minimum number of crossings can lead to more crossings than necessary when the next edge is inserted into the resulting graph. This is the same effect that occurs when the greedy method is applied to hard problems.

The average relative improvement was 14.42% in total. The maximum improvement was 85.71%. This happened for a graph with 39 vertices and 56 edges. The standard algorithm produced seven crossings whereas our new algorithm produced only one. The maximum negative improvement was -100% for a graph with 65 vertices and 76 edges. Here, the standard algorithm produced two crossings and the new algorithm four crossings. The average number of crossings produced by the standard algorithm grows from 1.29 for graphs with 11 vertices to 57.86 for graphs with 100 vertices, whereas the numbers grow from 1.29 to 49.83 for our new algorithm.

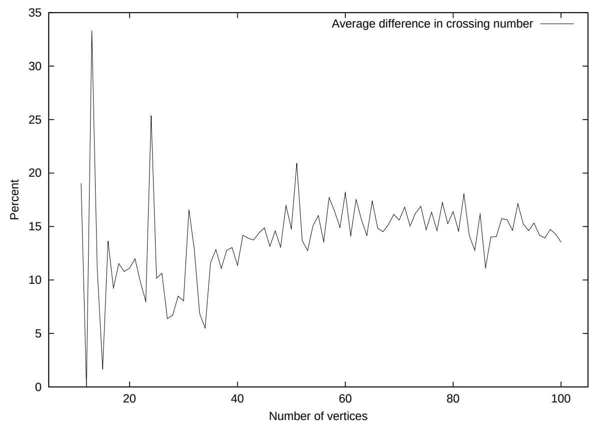


Figure 3.28: Average relative improvement for graphs with the same number of vertices when using our new method.

Since the asymptotic running time of the standard planarization method is the same as the asymptotic running time of our modified planarization method and the improvements using the modified version are significant, it seems well worth using our modified version instead of the standard version.

3.6 Crossing Number and Edge Re-Insertion

We have shown how to solve the following problem optimally: Given a planar graph $G = (V, E)$ and an additional edge e, find a drawing of $G' = (V, E \cup \{e\})$ that has the minimum number of crossings under the constraint that edges in E do not cross each other. This section shows that the number of crossings in this solution can be much higher than the number of crossings in a drawing of G' where the edges of E are allowed to cross.

To find such an example, we need a graph G' which is not planar but can be made planar by deleting just one edge. Since the number of edges in a graph that we have to delete to make it planar is called the *skewness* of a graph, our graph G' must have skewness one.

The example found by Farr (20) is shown in Figure 3.29(a). The grey areas in the graph are subgraphs that look like the graph in Figure 3.29(b). We call these graphs *walls*. The vertices v_1 and v_2 (drawn as hollow circles) are called the *poles* of a wall and are adjacent to a chain of n vertices (shown in grey). We call n the *thickness* of the wall. The first and the last vertex of this chain are connected to the poles via a simple path of arbitrary length. A wall has certain properties that are used in the example:

1. A wall is a subdivision of a triconnected graph and has therefore only two combinatorial embeddings that are mirror images of each other. We call a graph with this property *rigid*.

2. Assume that a graph contains a ring of walls, each with thickness n such that the poles of neighboring walls are identified. Then inserting an edge that starts at a node inside the ring and ends at a node outside of the ring produces at least n crossings.

The graph in Figure 3.29(a) contains a ring of walls (as in Figure 3.29(b), the hollow vertices are the poles of the walls). By making these walls thick enough, we can make sure that in any drawing with the minimum number of crossings, there are no edges that start inside the ring and end outside of the ring.

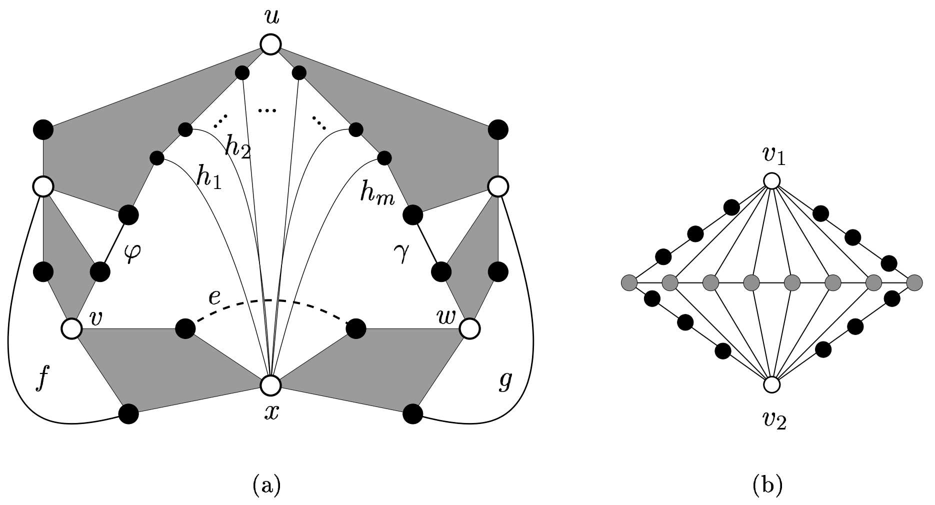


Figure 3.29: The graph G with skewness one on the left and an example of a wall with thickness eight on the right.

We define the graph $G = (V, E)$ as the graph shown in Figure 3.29(a) without the dashed edge e and the graph G' as $G' = (V, E \cup \{e\})$. We want to show that the drawing of G' with the minimum number of crossings where edges in E do not cross does not have the minimum number of crossings among all drawings of G'.

To show this, we have to define values for the parameter m (the number of non-wall edges incident to vertex x) and the thickness of the walls. We choose $m = 2 + p$ for some positive integer p. We choose the thickness of the walls in the ring as $(m + 5)^2 + 1$. This is the square of the number of edges in G' that are not contained in the walls plus 1 (the 5 edges are φ, γ, f, g, and e). This makes sure that even a drawing where all the non-wall edges cross each other has less crossings than any drawing where the ring of walls is crossed by an edge.

First we show that the drawing in Figure 3.29(a) has the minimum crossing number among all drawings of G' where edges of E do not cross. Each wall is rigid and the edges φ, γ, f, and g make sure that we cannot flip a wall around the axis going through its poles without producing crossings. Therefore the graph G' itself is rigid. Since G' is rigid, and crossing a wall produces too many crossings, the best way of inserting e is to cross the edges h_1 to h_m as shown in Figure 3.29(a). This results in m crossings.

Figure 3.30 shows a drawing of G' with only two crossings. This drawing is produced by flipping the two walls that connect the vertices v, x and w around their poles. Now edge f crosses edge φ while edge g crosses edge γ. So there are crossings between edges of E in this drawing while there are no crossings that involve edge e.

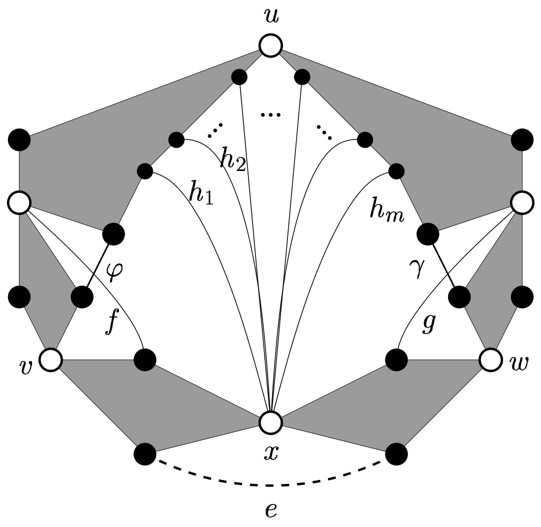


Figure 3.30: A drawing of graph G' with only two crossings

The parameter p is the difference of the number of crossings in Figure 3.29(a) and the number of crossings in Figure 3.30. Since we can choose p as any positive number, the difference in the number of crossings between the two drawings can be arbitrarily large.

This shows that inserting an edge into a planar graph, even if done in an optimal way, does not necessarily produce a drawing of the resulting graph with the minimum number of crossings. In fact, the difference to the optimal crossing number can be arbitrarily large.

Chapter 4

Computing Orthogonal Drawings with the Minimum Number of Bends

We study the problem of minimizing the number of bends in an orthogonal drawing over the set of all combinatorial embeddings of a given planar graph. The motivation for studying this problem is that the number of bends has a significant influence on the readability of an orthogonal drawing of a graph.

We characterize the set of all possible embeddings of a given biconnected planar graph G by means of a system of linear inequalities with $\{0, 1\}$-variables corresponding to the set of those cycles in G that can appear in a combinatorial embedding. This system of linear inequalities can be constructed recursively using SPQR-trees and a new splitting operation. To our knowledge, this is the first characterization of embeddings as an integer linear program. We published our findings in (42) and (43).

We also describe the concept of orthogonal representations and how an orthogonal representation with the minimum number of bends can be found for a given embedding by solving a minimum cost flow problem. We translate the minimum cost flow problem into a linear program and combine it with the integer linear program that describes all embeddings of a graph. This results in a mixed integer linear program that describes the orthogonal representations of a graph.

The algorithm we use to compute a drawing of a biconnected planar graph with the minimum number of bends involves computing the mixed integer linear program, finding an optimal solution, separating constraints and re-optimizing if necessary. We also present some methods we use to speed up the computation.

We present computational results on two different benchmark sets of graphs. There are two main results of these experiments. The first result is that our new approach is faster for large graphs with many embeddings than the previously known branch & bound approach for solving the bend minimization problem. The second result is that the use of exact algorithms reduces the number of bends significantly for almost half of the tested graphs.

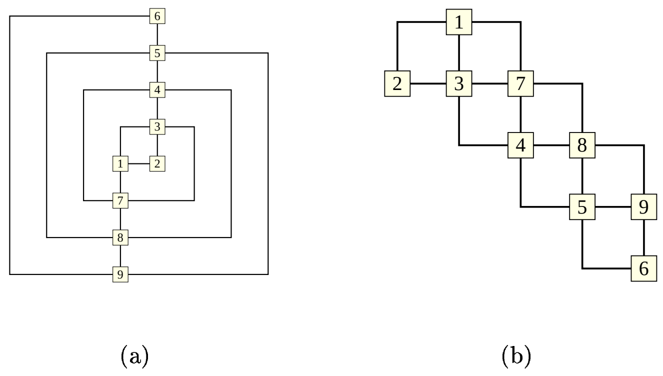


Figure 4.1: The impact of the chosen planar embedding on the drawing

4.1 Introduction to the Problem

The complexity of choosing an embedding of a planar graph has already received some attention (10; 11; 12). These authors have given polynomial time algorithms for computing an embedding of a planar graph that minimizes various distance functions on the faces of a graph with respect to the outer face like the radius, the width, the outerplanarity and the depth. Moreover, they have shown that computing an embedding that minimizes the diameter of the dual graph is NP-hard.

In this paper we deal with the following optimization problem concerned with embeddings: Given a planar biconnected graph, produce an orthogonal drawing of G that respects certain drawing conventions and has the minimum number of bends among all possible drawings of the graph that respect the conventions. This problem is NP-hard (24) and was already attacked in (7) and (9). The authors use a branch and bound approach, while we attack the problem using integer linear programming.

Our motivation to study this optimization problem and in particular an integer linear programming formulation arises in graph drawing. Most algorithms for drawing planar graphs need not only the graph as input but also a combinatorial embedding. The aesthetic properties of the drawing often change dramatically when a different embedding is chosen as the input for the algorithm.

Figure 4.1 shows two different drawings of the same graph that were generated using the bend minimization algorithm by Tamassia (47). The algorithm used different combinatorial embeddings as input. Drawing 4.1(a) has 13 bends while drawing 4.1(b) has only seven bends. Note that both drawings have the minimum number of bends among all drawings that realize the same combinatorial embedding. As the example indicates, it makes sense to look for the embedding that will produce the drawing with the smallest number of bends.

In graph drawing it is often desirable to optimize some cost function over all possible embeddings in a planar graph. In general these optimization problems are NP-hard. Garg and Tamassia (24) show, that minimizing the number of bends in an orthogonal planar drawing is

NP-hard. For a fixed planar embedding, this problem can be formulated as a flow problem in the geometric dual graph. Once we have characterized the set of all feasible embeddings (via an integer linear formulation on the variables associated with cycles), we can use this in an integer linear programming formulation for the corresponding flow problem. The variables of the resulting mixed integer linear program consist of 'flow variables' and 'embedding variables'.

In Section 4.2, we introduce an integer linear program whose set of feasible solutions corresponds to the set of all possible combinatorial embeddings of a given biconnected planar graph. One way of constructing such an integer linear program is by using the fact that every combinatorial embedding corresponds to a 2-fold complete set of circuits (see MacLane (40)). The variables in such a program are all simple cycles in the graph; the constraints guarantee that the chosen subset of all simple cycles is complete and that no edge of the graph appears in more than two simple cycles of the subset.

We have chosen another way of formulating the problem. The advantage of our formulation is that we only introduce variables for those simple cycles that form the boundary of a face in at least one combinatorial embedding of the graph, thus reducing the number of variables tremendously. Furthermore, the constraints are derived using the structure of the graph. We achieve this by constructing the program recursively using SPQR-trees. A new splitting operation makes the recursive construction possible.

The concept of orthogonal representations of planar graphs is defined in Section 4.3. Orthogonal representations describe the shape of orthogonal drawings of planar graphs without determining the length of the edge segments. We introduce different types of orthogonal representations that differ in the way they cope with high degree vertices. We present a minimum cost flow problem that is equivalent to finding an orthogonal representation with the minimum number of bends for a fixed embedding of a graph and formulate this problem as a linear program.

In Section 4.4, we combine the linear program for orthogonal representations with a fixed embedding and the integer linear program that describes the embeddings of a graph. This results in a mixed integer linear program where every optimum solution corresponds to an orthogonal representation with the minimum number of bends over all embeddings of the graph.

In Section 4.5, we present the algorithm that we use to compute a drawing of a planar biconnected graph with the minimum number of bends. This algorithm first computes an initial mixed integer linear program and solves it using a mixed integer solver. Then we check if the solution satisfies all the constraints that an orthogonal representation has to satisfy. If this is not the case, we add a violated constraint and compute a new solution.

When all constraints are satisfied, we translate the solution into an orthogonal representation. This representation is then taken as the input for an algorithm that computes an orthogonal drawing of the graph that realizes the representation.

Section 4.6 contains the computational results. They show that our new approach outperforms the previously known branch & bound method if the graph is large and has many embeddings. Another result is that compared with a heuristic often used for solving the problem, an exact solution saves a significant number of bends for almost half of the graphs we tested.

4.2 The Integer Linear Program Describing the Embeddings of a Graph

4.2.1 Intuition

The SPQR-tree represents the decomposition of a biconnected planar graph with respect to its triconnected components (see Chapter 2 for an introduction to SPQR-trees). All embeddings of the graph can be enumerated by enumerating all possible embeddings of these components. Our approach uses the fact that each skeleton of a node in the SPQR-tree represents a simplified version of the original graph. By computing the integer linear programs (ILP) for these simple graphs and using a lifting procedure, we can compute an ILP for the original graph.

When we take a closer look at the skeleton of a node in the SPQR-tree, we observe that it can be constructed from the original graph by replacing one or several subgraphs by single edges. Each edge in a skeleton connects two vertices that are a split pair in the original graph. We can think of such an edge as representing a subgraph of the original graph that connects the two nodes of the split edge. So every cycle in a skeleton represents a set of cycles in the original graph that we get by replacing edges of the cycle by paths in the subgraph that they represent.

The variables of our program correspond to directed cycles in the graph that are face cycles in at least one planar embedding. We can guarantee this, because our recursive construction computes the set of variables for the original problem using the sets of variables from subproblems for which the ILP has already been computed. So we construct cycles in the original graph from cycles in the subproblems by replacing certain edges with paths in the subgraphs that are represented by these edges.

4.2.2 The Variables of the Integer Linear Program

The skeletons of P-nodes are *multi-graphs*, so they have multiple edges between the same pair of nodes. Because we want to talk about directed cycles in these skeletons, we can be much more precise when we introduce *half edges*. We associate with each edge $e = (v, w)$ in the undirected graph $G = (V, E)$ two directed half edges $\vec{e}_1 = (v, w)$ and $\vec{e}_2 = (w, v)$. We say that $\vec{e}_1$ is the *reversal* of $\vec{e}_2$ and vice versa. We denote the reversal of $\vec{e}$ by $r(\vec{e})$. The set of half edges associated with the graph G is $\vec{E}$.

A *directed cycle* in the undirected graph $G = (V, E)$ is a sequence of half edges of the following form: $c = ((v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1)) = (\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_k)$ with the properties that every vertex of the graph is incident to at most two half edges of c and for each edge e of G, at most one of the half edges of e is contained in c. A *face cycle* in a combinatorial embedding of a undirected planar graph is a directed cycle of the graph, such that in any planar drawing that realizes the embedding, the right side of the cycle is empty. Note that the number of face cycles of a planar biconnected graph is $m - n + 2$ where m is the number of edges in the graph and n the number of vertices.

Now we are ready to construct an ILP, where the feasible solutions correspond to the combinatorial embeddings of a biconnected planar graph. The variables of the program are binary and they correspond to directed cycles in the graph, so every binary variable x_c corresponds to a directed cycle c. In a feasible solution of the ILP, a variable x_c has value 1 if the associated cycle is a face cycle in the represented embedding and 0 otherwise. To keep the number of variables as small as possible, we only introduce variables for those cycles that are a face cycle in at least one combinatorial embedding of the graph.

4.2.3 The Integer Linear Program for SPQR-trees with One Inner Node

We know that a graph whose SPQR-tree has only one inner node is isomorphic to the skeleton of this inner node. The ILPs for SPQR-trees with only one inner node are defined as follows, depending on the type of the node:

- S-node: When the only inner node of the SPQR-tree is an S-node, the whole graph is a simple circle. Thus it has two directed cycles and both are face-cycles in the only combinatorial embedding of the graph. So the ILP consists of two variables, both of which must be equal to one.

- R-node: In this case, the whole graph is triconnected. Every triconnected graph has exactly two combinatorial embeddings, which are mirror-images of each other. So the ILP we are looking for must have exactly two solutions.

  Each of the two embeddings of the graph has $m - n + 2$ faces. We get the face cycles of one embedding by reversing all the face cycles of the other embedding since the two embeddings are mirror images of each other. There is no cycle in the graph that is a face cycle in both embeddings. It follows that our ILP has exactly $k = 2(m - n + 2)$ variables. The constraints are given by the convex hull of the two points in k-dimensional space that correspond to the two embeddings.

  The computation of the cycles that are represented by the variables is easy: We just compute an arbitrary embedding of the graph, and take its face cycles. The face cycles of the other embedding are the cycles we get by reversing the face cycles of the first embedding.

  If we order the variables for the face cycles such that the variables with odd number represent the face cycles of the first embedding and the variables with even number represent the second embedding, the ILP has a very simple structure. Let the variables of the problem be x_1 to x_k (note that k is even since we have $2(m - n + 2)$ variables). Then the ILP describing the embeddings is the following:

$$\begin{aligned} x_{2i-1} + x_k &= 1 \quad \text{for} \quad 1 \leq i \leq k/2 \\ x_{2i} - x_k &= 0 \quad \text{for} \quad 1 \leq i \leq k/2 - 1 \\ x_k &\geq 0 \quad \text{for} \quad 1 \leq i \leq k \\ x_k &\leq 1 \quad \text{for} \quad 1 \leq i \leq k \end{aligned}$$

Note that this is a complete description of the polytope that describes the combinatorial embeddings of an R-node skeleton. There is no need to demand that the solutions are integral.

- P-node: The ILP for graphs whose only inner node in the SPQR-tree is a P-node is described in detail in the next section because this is where the connection to the asymmetric traveling salesman problem (ATSP) is used.

4.2.4 P-Node Skeletons and the ATSP

A P-node skeleton P consists of two vertices v_a and v_b connected by $k \geq 3$ edges and has therefore $(k-1)!$ different combinatorial embeddings. The reason is that there are exactly $(k-1)!$ different circular orderings of k objects. Every directed cycle in P is a face cycle in at least one embedding of P. Each ordered pair of edges in P represents a face cycle, so we need

$$2\binom{k}{2} = k^2 - k$$

variables in an ILP description of all combinatorial embeddings.

Let C be the set of variables in the ILP and $c : C \to \mathbb{R}$ a weight function on C. We consider the problem of finding the embedding of P that minimizes the sum of the weights of the cycles that are face cycles. We will show that this problem can be stated as the problem of finding a Hamiltonian Cycle with minimum weight in the complete directed graph with k vertices or simply as the *asymmetric traveling salesman problem* (ATSP). The transformation we will present here also works in the opposite direction thus showing NP-hardness of optimizing over all embeddings. But here we only show the reduction to ATSP because this is what our algorithm does when it computes the ILP.

The complete directed graph B has one vertex for every edge of P. Let e_1 and e_2 be two edges in P and v_1 and v_2 the corresponding vertices in B. Then the directed edge (v_1, v_2) in B corresponds to the cycle in P consisting of half edge $\vec{e}_{1.1} = (v_a, v_b)$ of e_1 and half edge $\vec{e}_{2.2} = (v_b, v_a)$ of e_2. The edge (v_2, v_1) in B corresponds to the same cycle in the opposite direction. It corresponds to the cycle consisting of half edge $\vec{e}_{1.2} = (v_b, v_a)$ and half edge $\vec{e}_{2.2} = (v_a, v_b)$. In this way, we define a bijection $b : C \to E$ from the cycles in P to the edges in B. This bijection defines a linear function $c' : E \to \mathbb{R}$ on the edges of B such that $c'(e) = c(b^{-1}(e))$.

Now it is not hard to see that each embedding of P corresponds to a Hamiltonian Cycle in B and vice versa. If the Hamiltonian Cycle is given by the sequence $(v_1, v_2, \ldots, v_k)$ of vertices, then the sequence of the edges in P in the corresponding embedding in clockwise order around v_a is $(e_1, e_2, \dots, e_k)$. Figure 4.2 shows an example of a P-node skeleton with four edges and the corresponding ATSP-graph. The embedding of the P-node skeleton on the left corresponds to the Hamiltonian Cycle marked by thick edges in the ATSP-graph. The marked edges correspond to the face cycles in the embedding of the P-node skeleton. The sequence of the edges in P in counter-clockwise order around vertex v_a corresponds to the sequence of the vertices in the Hamiltonian Cycle of the ATSP-graph.

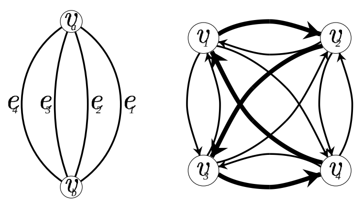


Figure 4.2: A P-node skeleton and its corresponding ATSP-graph

The sum of the weights of all edges on the Hamiltonian Cycle is equal to the sum of the weights of the cycles of P that are face cycles in this embedding. So finding an embedding of P that minimizes the sum of the weights of the face cycles is equivalent to finding a traveling salesman tour with minimum weight in B. Since we can easily construct a corresponding P-node embedding problem for any ATSP-problem, we have a simple proof that optimizing over all embeddings of a graph is in general NP-hard.

The equivalence of ATSP and the embedding problem for P-node skeletons enables us to use the same ILP used for ATSP for the description of all combinatorial embeddings of a P-node skeleton. The formulation for the ATSP ILP found in (14) has two types of constraints:

1. The *degree constraints* state that each vertex must have exactly one incoming edge and one outgoing edge in a feasible solution.

2. The *subtour elimination constraints* state that the number of edges with both endpoints in a nonempty subset S of the set of all vertices can be at most $|S| - 1$.

The number of degree constraints is linear in the number of edges in a P-node skeleton, while the number of subtour elimination constraints is exponential. We want to use the ILP for the ATSP problem to define the ILP that describes all combinatorial embeddings of a P-node skeleton. Since the number of subtour elimination constraints is exponential in the number of edges in the skeleton, we define the ILP as the set of degree constraints for the corresponding ATSP-problem.

To cope with the subtour elimination constraints, we store for each P-node skeleton the corresponding ATSP-graph. For each edge in the ATSP-graph we store the corresponding cycle in the P-node skeleton. During the recursive construction of the ILP that represents all combinatorial embeddings of a graph, the set of cycles represented by an edge in the ATSP-graph may change, therefore we update the set of corresponding cycles for each edge in the

ATSP-graph. By doing this, we make sure that we always know the list of cycles represented by an edge in the ATSP-graph. This is done in the same way as the construction of the lifted constraints in subsection 4.2.6.

When the construction of the recursive ILP is finished, we use a mixed integer programming solver to find an integer solution. Then we check if any subtour elimination constraint is violated. We do this by finding a minimum cut in the ATSP-graph for each P-node skeleton in the SPQR-tree. The weight of each edge in the ATSP graph is defined as the sum of the values of the variables representing the cycles associated with the edge. If the value of this minimum cut is smaller than one, we have found a violated subtour elimination constraint and add it to the ILP. As we will show in Section 4.6 where we describe computational results, separation of the subtour elimination constraints is rarely necessary.

4.2.5 Splitting an SPQR-Tree

We know that by fixing a combinatorial embedding of the skeleton of each node in the SPQR-tree, we fix a combinatorial embedding of the original graph. There are only two types of nodes in the SPQR-tree whose skeletons have more than one combinatorial embedding: the P-nodes and the R-nodes. The skeletons of S- and Q-nodes are simple circles and have only one combinatorial embedding. So the embeddings we choose for the skeletons of R- and P-nodes determine the embedding of the original graph. Therefore, these nodes will play an important part in the recursive construction of the ILP and we call them the *decision nodes* of the SPQR-tree.

We use a recursive approach to construct the variables and constraints of the ILP. Therefore, we need an operation that constructs a number of smaller problems out of our original problem such that we can use the variables and constraints computed for the smaller problems to compute the ILP for the original problem.

This is done by splitting the SPQR-tree at some decision node v. Let e be an incident edge of v whose other endpoint is not a Q-node. Deleting e splits the tree into two trees T_1 and T_2. We add a new edge with an attached Q-node to both trees to replace the deleted edge and thus ensure that T_1 and T_2 become complete SPQR-trees again. Note that every SPQR-tree with more than one inner node contains at least one decision node. The reason is that two S-nodes can never be adjacent. This follows from the construction of the SPQR-tree.

For adjacent edges of v whose other endpoint is a Q-node, the splitting is not necessary. Doing this for each edge incident to v results in $d+1$ smaller SPQR-trees, called the *split trees* of v, where d is the number of inner nodes adjacent to v. Remember that the Q-nodes are the leaves of an SPQR-tree and all other node types are inner nodes of the tree. Since the new trees are SPQR-trees, they represent planar biconnected graphs which are called the *split graphs* of v. We will show how to compute the ILP for the original graph using the ILPs computed for the split graphs.

Consider the graph shown in Figure 4.3 as an example for the splitting operation. Its SPQR-tree is shown in Figure 4.4. We choose the P-node P_1, shown in grey in the SPQR-tree, as the split node. Note that P_1 is a decision node of the SPQR-tree.

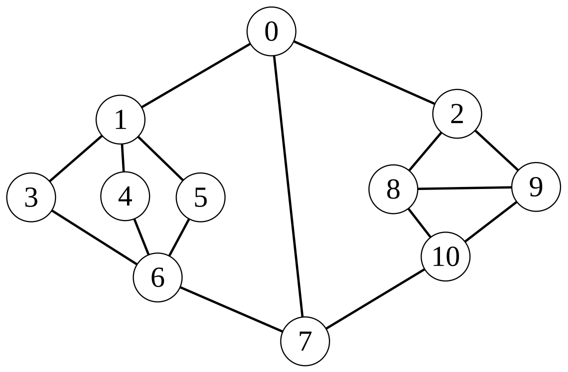


Figure 4.3: Example graph for the splitting operation.

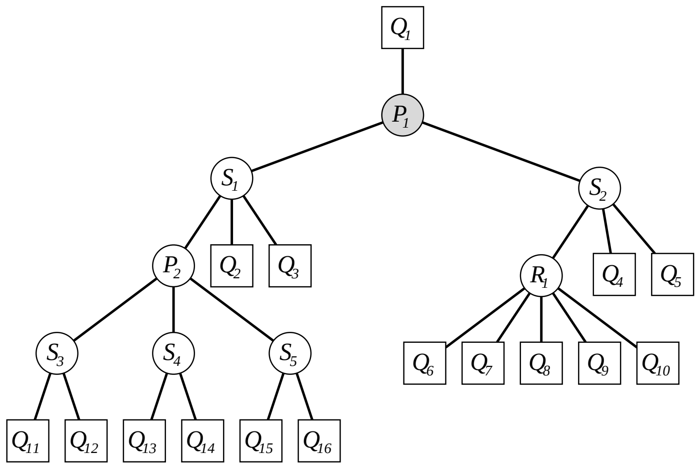


Figure 4.4: The SPQR-tree for the graph in Figure 4.3. The Q-nodes are drawn as rectangles and the split node is drawn in grey.

We get the split trees by removing the edges that connect P_1 to adjacent inner nodes of the tree and then introducing new Q-nodes to make sure that the resulting split trees are SPQR-trees. This results in the three trees shown in Figure 4.5. The added Q-nodes are drawn in grey. We call the tree that contains the split node the *center split tree*. This is the tree shown in Figure 4.5(a). There are two added Q-nodes in the tree and one Q-node that was already contained in the original tree (the node Q_1). The other two split trees shown in the Figures 4.5(b) and 4.5(c) have one added Q-node each. The splitting operation guarantees that each split tree except from the center split tree has exactly one added Q-node. The center split tree can have more than one.

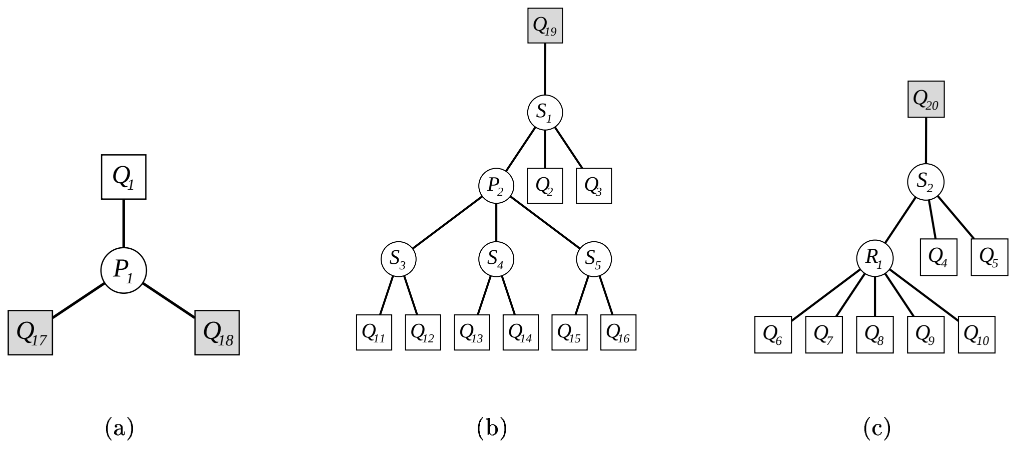


Figure 4.5: The split trees of the SPQR-tree in Figure 4.4 when P_1 was chosen as the split node. The added Q-nodes are drawn in grey.

Figure 4.6 shows the split graphs that belong to the three split trees shown in Figure 4.5. The edges that correspond to added Q-nodes are drawn as grey lines. We call these edges *split edges* because they are a result of the splitting process. Only the center split graph (the graph that belongs to the center split tree) can have more than one split edge while all other split graphs have exactly one split edge.

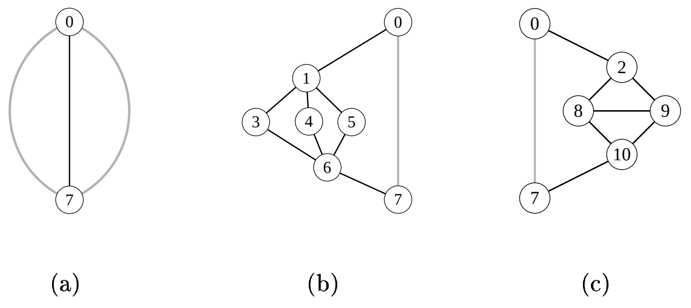


Figure 4.6: The split graphs of the graph in Figure 4.4 when P_1 was chosen as the split node. The split edges are drawn in grey.

As we have seen, the number and types of decision nodes in the SPQR-tree of a graph determines the number of combinatorial embeddings. The subproblems we generated by splitting the tree either have only one decision node or at least one less than the original problem. So by splitting the SPQR-tree, we have produced sub-problems that are easier to handle than the original problem.

4.2.6 Construction of the ILP for SPQR-trees with More Than One Inner Node

We define how to construct the ILP of an SPQR-tree $\mathcal{T}$ from the ILPs of the split trees of a decision node v of $\mathcal{T}$. Let G be the graph that corresponds to $\mathcal{T}$ and $T_1, \ldots, T_k$ the split trees of v representing the graphs G_1 to G_k. We assume that T_1 is the center split tree of v. Now we consider the directed cycles of G. We can distinguish two types:

1. *Local cycles* are cycles of G that also appear in one of the graphs $G_1, \ldots, G_k$.
2. *Global cycles* of G are not contained in any of the G_i. These are the cycles that have been split into several smaller cycles by the splitting operation.

Every split tree of v except for the center split tree is a subgraph of the original graph G with one additional edge (the *split edge* corresponding to the added Q-node). The graph that corresponds to the center split tree may have more than one split edge. Note that the number of split edges in this graph is not necessarily equal to the degree of v, because v may have been connected to Q-nodes in the original tree.

For every split edge e, we define *expand*(e) as the subgraph of the original graph G that is represented by e. The two nodes connected by a split edge always form a split pair p of G. When e is the split edge of the graph G_i represented by the split tree T_i, then *expand*(e) is the union of all the split components of G that share no edge with G_i. Consider for example the split graph shown in Figure 4.6(b). The grey edge is the split edge of this graph. If we define this edge as e, then *expand*(e) consists of the union of the black parts of the other split graphs. The reason is that the graphs in Figure 4.6 represent the three split components of the pair $\{0,7\}$ with the grey edges added. Building the union of the split components of $\{0,7\}$ that share no edge with the graph from Figure 4.6(b) is the same as merging the two graphs from the Figures 4.6(a) and 4.6(c) without the grey edges.

For every directed cycle c in a graph G_i represented by a split tree, we define the set $R(c)$ of *represented cycles* in the original graph. For local cycles c, we define $R(c) = \{c\}$. Let c be a cycle in a split graph that contains a half edge of a split edge. The cycle c' in G is in $R(c)$, if it can be constructed from c by replacing every half edge of a split edge $\vec{e} = (v, w)$ in c by a simple path of half edges in *expand*(e) from v to w. Consider for example the cycle $c = (0, 1, 3, 6, 7)$ given as a sequence of vertices in the split graph of Figure 4.6(b). It contains a half edge of the split edge $(0, 7)$. The set $R(c)$ contains the following cycles of the original graph in Figure 4.3:

$$\begin{aligned}
c_1 &= (0,1,3,6,7)\\
c_2 &= (0,1,3,6,7,10,9,2)\\
c_3 &= (0,1,3,6,7,10,8,2)\\
c_4 &= (0,1,3,6,7,10,8,9,2)\\
c_5 &= (0,1,3,6,7,10,9,8,2)
\end{aligned}$$

Each variable x_c in the ILP of a split tree that represents a local cycle c will also be a variable of the ILP of the original graph G. But we will also have variables that correspond to global cycles of G. A global cycle c in G will get a variable in the ILP, if the following conditions are met:

1. There is a variable x_{c_1} in the ILP of T_1 (the center split tree) with $c \in R(c_1)$.

2. For every split tree T_i with $2 \leq i \leq k$ where c contains at least one half edge of G_i, there is a variable x_{c_i} in the ILP of T_i such that $c \in R(c_i)$.

Now we have defined all the variables for the ILP of G. The set I of all constraints of the ILP is given by

$$I = I_l \cup I_C \cup I_G \ .$$

First we define the set I_l which is the set of *lifted constraints* of the ILP. Each of the graphs $G_1, \ldots, G_k$ is a simplified version of the original graph G. They can be generated from G by replacing some split components of one or more split pairs by single edges. When we have a constraint that is valid for a split graph, a weaker version of this constraint is still valid for the original graph. The process of generating these new constraints is called *lifting* because we introduce new variables that cause the constraint to describe a higher dimensional half space or hyper plane.

Let

$$\sum_{j=1}^{l} a_j x_{c_j} \doteq R$$

be a constraint for a split graph, where $\doteq \in \{\leq, \geq, =\}$ and C_x be the set of cycles in G that are represented by variables in the ILP of G. Then the lifted constraint for G is the following:

$$\sum_{j=1}^{l} a_j \sum_{c_i \in R(c_j) \cap C_x} x_{c_i} \doteq R$$

We define I_l as the set of lifted constraints of all the split graphs. The number of constraints in I_l is the sum of all constraints in all split graphs.

The set I_C is the set of *choice constraints.* Let c be a cycle in the graph with $|R(c)| > 1$. All the cycles in $R(c)$ either share at least one half edge or they pass a split graph in the same direction. Therefore, only one of the cycles in $R(c)$ can be a face cycle in any combinatorial embedding of G. Consider the two directed cycles in Figure 4.7. One is drawn in black while the other is drawn in grey. Each pair of the vertex set $\{0, 3, 6\}$ forms a split pair of the graph. The two cycles pass all these split pairs in the same direction. It follows that they can never be face cycles in the same embedding. In the embedding realized by the drawing, the black cycle is a face cycle, while the grey cycle is not. The cycle that contains the reversals of the half edges of the grey cycle is a face cycle and forms the boundary of the central face.

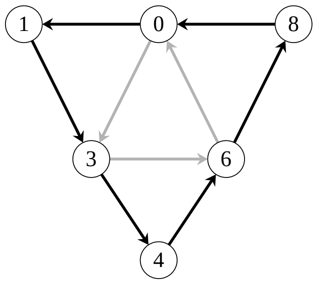


Figure 4.7: Two cycles in a graph that can never be face cycles in the same embedding.

For each variable x_c in the ILP of a split graph with $|R(c)| > 1$ we therefore have one constraint that has the following form:

$$\sum_{c' \in R(c) \cap C_x} x_c \leq 1$$

The set I_G consists of only one constraint, called the *center graph constraint*. Let F be the number of face cycles in a combinatorial embedding of G_1 (the center split graph), C_G the set of all global cycles c in G and C_l the set of all local cycles in G_1. Then this constraint is:

$$\sum_{c \,\in\, (C_G \cup C_l) \cap C_x} x_c = F$$

This constraint is valid, because we can produce every drawing D of G by replacing all split edges in a drawing D_1 of G_1 with the drawings of subgraphs of G. For each face cycle c_1 in D_1, there will be a face cycle c in D, such that c and c_1 are either identical (if it was a local cycle) or c is a global cycle. This defines the ILP for any biconnected planar graph.

4.2.7 Correctness

The proof of correctness works recursively like the construction of the ILP. We first proof that the ILP is correct for the basic graphs and then show that it is correct for more complex graphs.

Because the proof of the main theorem is quite complex, we split it into two lemmas. We first define the *incidence vector* of a combinatorial embedding. Let C be the set of all directed cycles in the graph that are face cycles in at least one combinatorial embedding of the graph. Then the incidence vector χ_Π of Π is given as a vector in $\{0,1\}^{|C|}$ where the components representing the face cycles in Π have value one and all other components have value zero.

In the following lemmas, we assume that the subtour elimination constraints are treated like any other constraint. So we assume that they are included in the ILP for P-nodes and that they are lifted when the ILP for larger graphs is computed. We do not do this in our implementation because the number of constraints would grow too quickly with the size of the graph. Instead, we do not include the subtour elimination constraints in the ILP and check for every computed solution if it violates any lifted subtour elimination constraint using the ATSP-graphs computed for each P-node skeleton (see Section 4.2.4 on page 74). If we find no violated lifted subtour elimination constraint, the solution is also valid for the ILP where all lifted subtour elimination constraints are present. Otherwise, we add the violated subtour elimination constraint that we have found to the ILP and re-optimize until there are no more violated subtour elimination constraints. So we can assume in the following proofs that all subtour elimination constraints are treated like the rest of the constraints and added to the ILP from the beginning.

Lemma 4.1 *Let $C_\Pi = \{c_1, c_2, \ldots, c_k\}$ be the set of cycles that are face cycles in a combinatorial embedding Π of the biconnected planar graph G. Then the incidence vector χ_Π satisfies all constraints of the ILP we have defined.*

Proof We prove the lemma using induction over the number n of inner nodes in the SPQR-tree $\mathcal{T}$ of G. The value $\chi(c)$ is the value of the component in χ associated with the cycle c. We do not consider the case $n = 0$, because G is a simple cycle with two vertices and two edges in this case and has only one combinatorial embedding.

$n = 1$: No splitting of the SPQR-tree is necessary, the ILP is defined directly by $\mathcal{T}$. Inspection of the three different cases in Section 4.2.3 on page 73 verifies that χ satisfies all constraints.

$n > 1$: Let μ be the node in $\mathcal{T}$ we used in the construction of the ILP to split $\mathcal{T}$ into the split trees $T_1, \ldots, T_d$. We split C_Π into two subsets C_G and C_L such that C_G holds all global cycles of C_Π and C_L the local cycles of C_Π with respect to μ (the sets of global and local cycles depend on the choice of the split node). Then every cycle c in C_L is contained in some split graph G_i of μ with $1 \leq i \leq d$.

The combinatorial embedding Π uniquely defines a combinatorial embedding Π_i for each G_i. We define C_{Π_i} as the set of face cycles in Π_i. Every cycle in C_{Π_i} that does not contain a half edge of the split edge of G_i, is also contained in C_Π. Apart from local cycles that are contained in C_Π, every C_{Π_i} with $2 \leq i \leq d$ contains two more cycles that are not contained in C_Π. This is true because every G_i has exactly one edge that is not contained in G: the split edge of G_i. In every embedding of a biconnected planar graph, the two half edges of each edge are contained in different cycles, so there are two cycles in C_{Π_i} that are not cycles in C_Π.

Consider now the choice constraints of the ILP for all cycles c in a split graph G_i of μ that include at least one split edge.

$$\sum_{c' \in R(c) \cap C_x} x_{c'} \leq 1$$

As we have already pointed out in the definition of the choice constraints, any two cycles in R_c share at least one half edge of G or pass at least one split graph of μ in the same direction. So they cannot be face cycles in the same combinatorial embedding. Since Π is a combinatorial embedding, χ_Π must satisfy all choice constraints of the ILP.

We use this fact and the induction basis to prove that χ_Π satisfies all lifted constraints. Because our claim holds for $n-1$, we know that the incidence vector χ_{Π_i} satisfies all constraints of the ILP for G_i with $1 \leq i \leq d$. The lifted constraints of the ILP are constructed out of the constraints for the G_i by replacing each variable x_c in the constraint by the sum of the variables for G that are associated with cycles in $R(c)$. The right side of the constraint and the relation sign remain unchanged. So we have to show that the left side of the constraint has the same value when we apply it to χ_{Π_i} as when we lift it and apply it to χ_Π. Because the choice constraints hold, this is equivalent to the following claim:

$$\chi_i(c) = \sum_{c' \in C_x \cap R(c)} \chi(c') \quad \forall \text{ cycles } c \text{ in } G_i \text{ represented by a component of } \chi_i.$$

If c is a local cycle of G, this is obvious, because we have $R(c) = \{c\}$ and c is a face cycle in Π if and only if it is a face cycle in Π_i. Now we assume that c contains a half edge of a split edge of G_i. Every split edge in G_i represents a subgraph of G, and each face cycle of Π_i was generated from a face cycle of Π by collapsing some path into a single half edge. So when $\chi_i(c)$ is zero, there can be no face cycle in Π that is in the set $R(c)$ and when $\chi_i(c)$ is one, there must be exactly one cycle in $R(c)$ that is a face cycle of Π. So the left side of the constraint has the same value when we lift it and apply it to χ_Π. It follows that the constraint is satisfied.

The local cycles in G_1 are the cycles that consist only of edges contained in G. When l is the set of local cycles contained in G_1, C_G the set of global cycles in G, and F the number of face cycles in any combinatorial embedding of G_1, then the center graph constraint has the following form:

$$\sum_{c \in (C_G \cup C_l) \cap C_x} x_c = F$$

The graph G_1 is the skeleton of the split node. So if c is a face cycle of Π and belongs to C_l, then it is also a face cycle of Π_1. And if c is a global face cycle of Π, the cycle c' in G_1 that was generated by replacing some path of half edges in c by a single half edge is a face cycle in Π_1.

On the other side, every face cycle of Π_1 is also a face cycle of Π, if it contains no split edge. For every face cycle c of Π_1 that contains at least one split edge, there is a global face cycle in Π that can be constructed from c by replacing the split edges with paths in the corresponding split graphs. Since the number of face cycles in Π_1 is F, the equality is satisfied by χ_Π.

□

Lemma 4.2 *If G is a biconnected planar graph and $\chi \in \{0,1\}^{|C|}$ a vector that satisfies all constraints of the ILP, then χ is the incidence vector of a combinatorial embedding Π of G.*

Proof Again, we use induction over the number n of inner nodes in the SPQR-tree $\mathcal{T}$ of G and we disregard the case $n = 0$ because G is only a simple cycle with two vertices in this case.

$n = 1$: $\mathcal{T}$ has only one inner node, then it is easy to verify our claim by checking the three cases in Section 4.2.3 on page 73.

$n > 1$ The proof works in two stages: First we construct vectors χ_i for each split graph G_i out of χ and prove that these vectors satisfy the ILPs for the corresponding split trees T_i. Therefore, the basis of the induction guarantees that each χ_i is the incidence vector of an embedding Π_i of G_i. In the second stage, we use the Π_i to construct an embedding Π for G and show that χ is the incidence vector of Π.

So first we construct the vectors χ_i. Let c be a cycle in some G_i that is a face cycle in at least one embedding of G_i. We set:

$$\chi_i(c) = \sum_{c' \in C_x \cap R(c)} \chi(c') \tag{4.1}$$

Because the choice constraints are satisfied and all components of χ are either zero or one, the value of this sum is always zero or one and so all components in χ_i will be either zero or one.

To show that χ_i is the incidence vector of a combinatorial embedding Π_i, it is sufficient to prove that χ_i satisfies all constraints of the ILP for T_i, because the induction basis guarantees that χ_i is an incidence vector of an embedding when all constraints are satisfied.

We know that χ satisfies all constraints that are lifted from constraints in the ILP for T_i. The right sides of the constraints do not change during lifting. Consider the left side of an arbitrary constraint lifted from the constraints of G_i, when C_i is the set of cycles in G_i that are associated with variables in the ILP for G_i:

$$\sum_{c \in C_i} (\lambda_c \cdot \sum_{c' \in R(c) \cap C_x} x_{c'}) \quad (\lambda_c \in \mathbb{R})$$

The left side of the original constraint in the ILP for G_i is:

$$\sum_{c \in C_i} (\lambda_c \cdot x_c)$$

We constructed χ_i such that $\chi_i(c) = \sum_{c' \in C_x \cap R(c)} \chi(c')$ and thus we know that all constraints of G_i are satisfied by χ_i and that it is the incidence vector of a combinatorial embedding Π_i of G_i.

We have completed stage one of the proof and proceed to stage two, so we want to construct an embedding Π for G from the Π_i and show that χ is the incidence vector of Π. We construct Π as described in Theorem 2.3 on page 32. So every cycle c in G_i for $1 \leq i \leq d$ which does not include a split edge is a face cycle in Π if and only if it is a face cycle in Π_i. Consider the way we have defined each χ_i in equation (4.1). Since c does not include a split edge, we have $R(c) = \{c\}$. And so we have that $\chi(c) = 1$ if and only if c is a face cycle in Π. We have now proven that χ and Π agree about the local cycles. We still have to show that they agree about the global cycles.

First we show that the number of components with value one in χ that are associated with global cycles is equal to the number of global cycles in the set C_Π (the set of face cycles in Π). This is guaranteed by the center graph constraint:

$$\sum_{c \in (C_G \cup C_l) \cap C_x} x_c = F$$

Here C_G is the set of all global cycles and C_l the set of all local cycles in G_1 (the center split graph). F is the number of face cycles in an embedding of G_1. We have shown in the previous lemma that this constraint is valid for every embedding of G. Since χ and Π agree on all local cycles, the number of global cycles whose associated component in χ is one is $F - |C_l \cap C_\Pi|$ which is equal to the number of global cycles in Π. We will now show that for all $c_g \in C_G \cap C_x$ we have $\chi(c_g) = 1$ and thus prove the lemma.

Let c_g be any cycle in $C_G \cap C_\Pi$. Then there is exactly one cycle c'_g in the center split graph G_1 with $c_g \in R(c'_g)$. We can construct c'_g from c_g by replacing certain paths with single edges. We have $c'_g \in C_{\Pi_1}$ (the set of face cycles in the embedding Π_1 of the center split graph G_1) which follows from Theorem 2.3 on page 32. From that we can conclude that there is a cycle c''_g in G with $c''_g \in R(c'_g)$ such that $\chi(c''_g) = 1$ (from the construction of χ_1 and Π_1). So we have to show that $c''_g = c_g$ holds.

Since $\{c''_g, c_g\} \subseteq R(c'_g)$, both cycles are represented by the same cycle c'_g in G_1. c'_g contains split edges and might also contain edges from G. When A_1 is the set of edges in c'_g that are also edges in G and A_2 the set of split edges in c'_g, then the edges in A_1 must be present in c_g and c''_g. So both cycles can only differ in the paths they take through the split graphs represented by the split edges in A_2.

Let e be an edge in A_2 and G_i be the split graph associated with this split edge. When p is the path of c_g that runs through G_i, then there is a face cycle c_l in Π_i that consists of p and a half edge of the split edge of G_i. This follows from the construction of Π. If c''_g took a path p' different from p through G_i, we would have another face cycle in Π_i that includes the same half edge of the split edge as c_l. But in any embedding of G_i, there is only one cycle that contains any given half edge of G_i. Therefore, p and p' must be the same because the embedding of G_i determines the path of every global cycle through G_i.

Since we can apply this argument to all split edges in the cycle c'_g, we can show that c_g and c''_g are in fact the same cycle. Thus we have $\chi(c_g) = 1$ and it follows that χ is indeed the incidence vector of the combinatorial embedding Π.

□

It may seem strange that the only constraints we explicitly mention in the proof of the lemma are the choice constraints, the lifted constraints and the center graph constraints. The reason is that all other constraints are computed once for split graphs that have only one inner node in their SPQR-tree and then lifted when the ILPs for the more complex subgraphs are computed.

The only new constraints that are computed during the composition of the ILP for a complex graph from the ILPs for its split graphs are the choice constraints and the center graph constraint. We can think of them as the constraints that form the connection between the ILPs computed for the subgraphs.

Now the two Lemmas 4.1 and 4.2 together imply the main theorem of this section. This theorem says that the ILP we have defined to describe all combinatorial embeddings of a planar biconnected graph is correct.

Theorem 4.1 *Every feasible solution of the generated ILP is the incidence vector of a combinatorial embedding of the given biconnected planar graph G and vice versa: The incidence vector of every combinatorial embedding of G corresponds to a feasible solution for the generated ILP.*

4.3 The Linear Program Describing Orthogonal Representations for a Fixed Embedding

4.3.1 Intuition

In this section, we present a linear program that can be used to compute an orthogonal representation with a minimum number of bends for a planar biconnected graph and a fixed embedding. This linear program is based on a minimum cost flow formulation of the same problem. So we will first introduce the flow network where a minimum cost flow corresponds to an orthogonal representation with the minimum number of bends. But before we do that, we have to define the concept of an orthogonal representation.

4.3.2 Orthogonal Representations

An orthogonal drawing of a graph is a drawing where all edges are sequences of vertical and horizontal line segments. The points where two segments of the same edge meet are called *bends*. Figure 4.8 shows an orthogonal drawing with four bends.

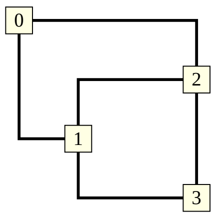


Figure 4.8: An orthogonal drawing with four bends.

Orthogonal representations are equivalence classes of planar drawings, just as planar embeddings. The difference is that they are equivalence classes only for orthogonal planar drawings of a graph and that they determine not only the topology of the drawing. An orthogonal representation additionally defines the sequence of the bends on each edge and the angle between each edge and its successor in the circular list of incident edges at a vertex defined by an embedding. An orthogonal representation always defines a planar embedding but there is an infinite number of orthogonal representations for each planar embedding. This is true because we can draw every graph with an arbitrary high number of bends.

An orthogonal representation does not define the length of the segments that connect the vertices and bends in the drawing. Therefore, there is an infinite number of different drawings that realize a certain orthogonal representation. The reason is that we can always stretch a drawing to make it cover an arbitrary large drawing area.

Note that two different drawings that realize the same orthogonal representation cannot always be constructed from each other by scaling the length of the edge segments. In fact, the drawings can look very different and it is in general NP-complete to decide for an orthogonal representation if it can be drawn on a grid with a fixed area (46).

The process of computing a drawing from an orthogonal representation of a graph is known as *compaction*. The PhD thesis of Klau (37) gives a good overview of the state of the art in this area. Klau, Klein, and Mutzel (38) present a comparison of different compaction algorithms.

First we define general orthogonal representations. The following definition is the same as the one used in the AGD-library (AGD) because we used this library for the implementation of our algorithms. Let $G = (V, E)$ be a planar graph and Π a planar embedding of G. Remember that a planar embedding is a combinatorial embedding where one face is chosen as the outer face. Let F be the set of faces of Π and $f_o \in F$ the outer face. Then an orthogonal representation is defined as a triple $\Omega = (\Pi, a, b)$ where a and b are functions on the half edges of G. The function a determines the angles between edges incident to the same vertex and b determines the bends on the edges.

$$\begin{aligned} a: \quad & \vec{E} \to \{0, 90, 180, 270, 360\} \\ b: \quad & \vec{E} \to \{0,1\}^* \end{aligned}$$

The two functions must have certain properties. To make the description of these properties easier, we introduce new terminology. Let $s = (s_1, \ldots, s_k) \in \{0,1\}^*$ be a string of zeroes and ones. We define $\bar{s}$ as follows: $\bar{s} = ((1 - s_k), \ldots, (1 - s_1))$. So we get the string $\bar{s}$ by replacing the zeroes with ones and vice versa and then reversing the sequence of the elements.

The function $z : \{0,1\}^* \to \mathbb{N}_0$ counts the number of zeroes in a string and the function $o : \{0,1\}^* \to \mathbb{N}_0$ the number of ones. The set $B(f)$ is the set of half edges on the boundary of face f. So the edges in $B(f)$ form a directed cycle such that f is on the right of each edge in $B(f)$. This enables us to give a compact description of the properties that the pair of functions a and b must have to qualify as an orthogonal representation. This description is taken from the paper of Tamassia (47). The same paper also contains the theorem that the functions a and b can be realized in an orthogonal drawing if and only if they satisfy the four conditions given below.

$$\sum_{\vec{e}=(v,u)\in\vec{E}} a(\vec{e}) = 360 \quad \forall u \in V \tag{4.2}$$

$$b(\vec{e}) = \overline{b(r(\vec{e}))} \quad \forall \vec{e} \in \vec{E} \tag{4.3}$$

$$\sum_{\vec{e}\in B(f)} (z(\vec{e}) - o(\vec{e}) + 2 - a(\vec{e})/90) = 4 \quad \forall f \in F - \{f_o\} \tag{4.4}$$

$$\sum_{\vec{e}\in B(f_o)} (z(\vec{e}) - o(\vec{e}) + 2 - a(\vec{e})/90) = -4 \tag{4.5}$$

Now we give an intuition what the functions a and b mean in the drawings that realize the orthogonal representation. The value $a(\vec{e})$ with $\vec{e} = (u, v)$ is the angle between the last segment of edge e (the segment that ends in v) and the first segment of the next half edge in the same face cycle as $\vec{e}$ (this segment starts in v). Note that each half edge of an embedded graph is on the boundary of exactly one face in each embedding. The string $b(\vec{e})$ defines the bends on half edge $\vec{e}$. A zero denotes a 90 degree bend to the right while a one denotes a 90 degrees bend to the left (with respect to the direction of $\vec{e}$).

Table 4.1 gives the values of the functions a and b for each half edge of the orthogonal representation that is realized by the drawing in Figure 4.9. As usual, we assume that the face bounded by a half edge is the face to the right of the half edge. Consider half edge $(3, 1)$ on the boundary of face F_1. The value of the function b for this half edge is (1,0,0,1,0) because the first bend on the half edge is a left bend followed by two right bends, a left bend and a final right bend. The value of function a for the half edge is 90, because the angle between the last segment of $(3, 1)$ and the first segment of the next half edge on the boundary of F_1 (the half edge $(1, 2)$) is 90 degrees.

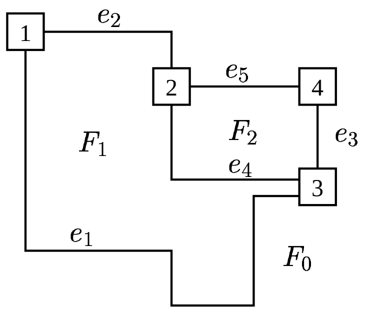


Figure 4.9: Example drawing for an orthogonal representation.

So now we can interpret the properties 4.2 to 4.5 in terms of the shape of a drawing that realizes the orthogonal representation. Equality 4.2 says that the sum of the angles around each vertex is 360 degrees. Equality 4.3 says that the bends on each half edge must agree with the bends on the reversal half edge. Equality 4.4 states that the sum of the angles in each inner face must be 360 degrees. This means that the face can be drawn as a closed shape with right angles. Equality 4.5 is the corresponding property of the outer face boundary.

half edge	$(1,3)$	$(3,1)$	$(1,2)$	$(2,1)$	$(2,3)$
a	270	90	180	270	0
b	(1,0,1,1,0)	(1,0,0,1,0)	(0)	(1)	(1)

half edge	$(3,2)$	$(2,4)$	$(4,2)$	$(3,4)$	$(4,3)$
a	90	90	90	270	90
b	(0)	()	()	()	()

Table 4.1: Values of the functions a and b for the orthogonal representation realized by the drawing of Figure 4.9.

The values in Table 4.1 satisfy all these constraints and so there must be a drawing of the corresponding graph that realizes the orthogonal representation given by the table. This is the orthogonal representation realized by the drawing shown in Figure 4.9.

In our approach, we use a special kind of orthogonal representations that we call *simplified orthogonal representations* . They were already used by Bertolazzi, Battista, and Didimo (9) and we have chosen these special orthogonal representations because the formulation of an orthogonal representation as a flow in a network is much easier than for general orthogonal representations.

Simplified orthogonal representations satisfy two additional constraints:

1. If a vertex has degree greater than four, then the largest angle between two incident edges is 90 degrees. So if the vertex is drawn as a square, at least one edge is incident to each side of the square.

2. Let $\vec{e}_1 = (w, v)$ and $\vec{e}_2 = (x, v)$ be two half edges such that $\vec{e}_2$ follows $\vec{e}_1$ in the circular sequence of incident edges of v in embedding Π. If the angle between $\vec{e}_1$ and $\vec{e}_2$ is zero degrees, then the following two conditions hold:

   (a) There is at least one bend on $\vec{e}_2$.

   (b) The last bend on $\vec{e}_2$ when traveling from x to v is a bend to the left.

These properties are satisfied in the orthogonal representation realized by the drawing shown in Figure 4.9. The only zero degree angle is at vertex 3. Half edge $(2, 3)$ follows half edge $(1, 3)$ in the clockwise circular sequence around vertex 3 and the last bend on $(2, 3)$ is a left bend.

4.3.3 Drawings of Orthogonal Representations

As we have already mentioned in the last section, the process of computing a drawing from an orthogonal representation is called compaction. There are several styles of orthogonal drawings. They differ in the approach they use to deal with vertices that have degree greater than four. This is a problem because if all vertices are drawn as points on the grid, then we cannot draw all edges as paths on this grid if there are more than four edges incident to a vertex. In this section, we present several styles of orthogonal drawings and an example drawing for each of them. We use the graph in Figure 4.10 as our example graph.

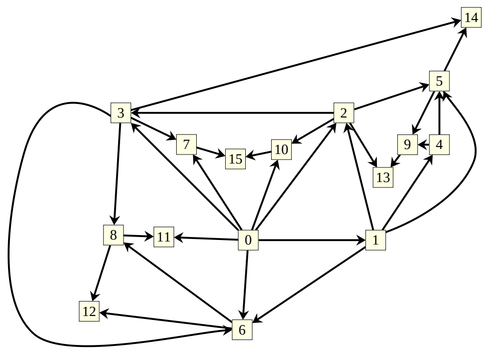


Figure 4.10: The example graph for the different orthogonal drawing styles.

One solution is to draw the vertices with degree greater than four not as points on the grid but as rectangles that cover several grid points. The size of the drawing of a vertex depends on its degree. Figure 4.11 shows an example for such a drawing. The advantage of this approach is that the edges are evenly spaced and that these drawings usually have a small number of bends. The disadvantage is that the size of the drawing of a vertex depends on its degree.

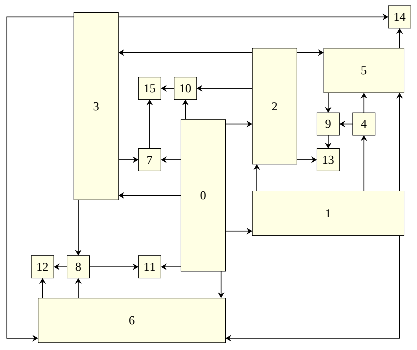


Figure 4.11: An orthogonal drawing of the graph in Figure 4.10 where the size of the drawing of a vertex depends on its degree.

Another approach is called the *podevsnef* approach. This is an abbreviation for “planar orthogonal drawings with equal vertex size and nonempty faces” and was introduced by Fößmeier and Kaufmann (22). As the name suggests, all the vertices are drawn as shapes (mostly squares) of the same size. To draw vertices with degree greater than four, the edges and bends are placed on a finer grid than the vertices. The term “empty face” refers to the drawing of a face such that we cannot fit a square of $1 - \varepsilon$ units width (the units are the grid units of the vertex grid) inside the drawing of the face.

Figure 4.12(a) shows an example for an empty face which is not allowed in a podevsnef drawing. It is not possible to fit a square of width $1 - \varepsilon$ units into the drawing of the inner face. Figure 4.12(b) shows the same graph drawn in the podevsnef model. In this drawing, we can even fit two squares of width $1 - \varepsilon$ into the drawing of the inner face. Figure 4.13 shows a drawing of the graph in Figure 4.10 in the podevsnef model.

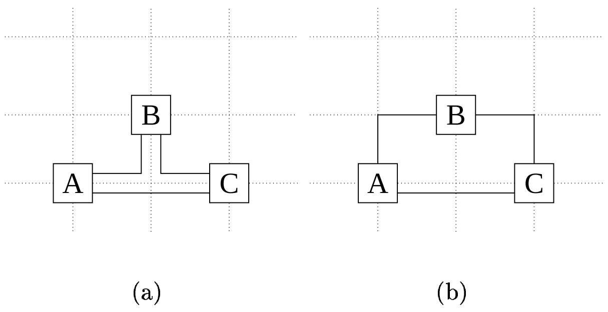


Figure 4.12: The drawing of an empty face (Figure 4.12(a)) and the same graph drawn in the podevsnef model (Figure 4.12(b).

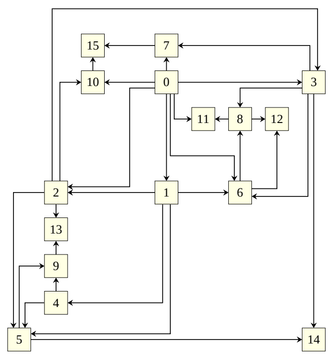


Figure 4.13: A podevsnef drawing of the graph in Figure 4.10.

The model for orthogonal drawings described by Bertolazzi, Battista, and Didimo (9) is called the *simple podevsnef model.* It only produces drawings for simplified orthogonal representations. The first difference to general podevsnef drawings is the following: When a set of edges runs parallel on neighboring grid points of the finer edge grid, then the rightmost edge bends away to the right from the parallel edges. We never have the case that an edge bends away to the left from a set of parallel edges. Figure 4.14 shows an example of a simple podevsnef drawing of the graph in Figure 4.10. The drawing in Figure 4.13 is not a simple podevsnef drawing. The edge $(3, 8)$ runs parallel to edge $(0, 3)$ but then bends to the left. This does not happen in Figure 4.14.

The second difference is that all vertices with degree greater than four have at least one incident edge on each side of the vertex. Again, this is not the case for general podevsnef drawing. In Figure 4.13, vertex 6 has degree five but there is no edge incident to the bottom of the square that represents vertex 6 in the drawing.

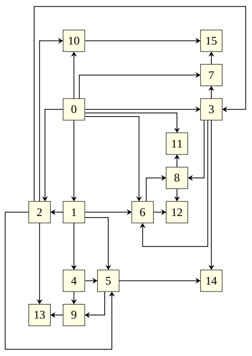


Figure 4.14: A simple podevsnef drawing of the graph from Figure 4.10.

The properties that distinguish general podevsnef drawings from simple podevsnef drawings correspond to the properties that distinguish a general orthogonal representation from a simplified orthogonal representation. An algorithm for drawing a general orthogonal representation can also be used to draw a simplified orthogonal representation and will produce a simple podevsnef drawing. This is in fact what we did in our implementation, because there is already an algorithm in the AGD-library (AGD) for drawing general orthogonal representations.

4.3.4 Describing an Orthogonal Representation by a Flow Network

Since Tamassia presented the first flow based algorithm for producing orthogonal drawings of graphs (47), network flows have been used in various algorithms in the area of orthogonal graph drawing (see for example Garg and Tamassia (25); Fößmeier and Kaufmann (22) and Bertolazzi, Battista, and Didimo (9)). While the approach used by Tamassia is designed to handle only graphs where the maximum degree of a vertex is four, the algorithms by Fößmeier and Kaufmann (22) and Bertolazzi, Battista, and Didimo (9) can handle graphs without restrictions on the degree of the vertices.

We have chosen the flow network presented by Bertolazzi, Battista, and Didimo (9) in our approach. The reason is that unlike the approach in Tamassia (47), it can deal with vertices of arbitrary degree and the flow network is much simpler than the one by Fößmeier and Kaufmann (22), because it computes simplified orthogonal representations.

The network $N = (V_N, E_N)$ is defined for a planar graph $G = (V, E)$ and a planar embedding Π of G with face set F and outer face $f_o \in F$. The edges of N are called *arcs* and the vertices *nodes* to avoid confusion with the edges and vertices of the original graph. The arcs of N have integer costs and capacities. We denote the cost of an arc e with $cost(e)$ and the capacity by $cap(e)$. Each node v in N supplies a certain amount of flow that we denote by $sup(v)$. If this value is negative, it is in fact a demand.

In the following definition, we say that the two faces f_1 and f_2 are adjacent, if there is a half edge $\vec{e}$ on the boundary of f_1 with the property that $r(\vec{e})$ is on the boundary of f_2. We define the degree $deg(f)$ of a face f as the number of half edges on its boundary.

We define the set $L(V)$ as the set of all vertices in V whose degree is at most four and $H(V)$ as the rest of the vertices in V. We call the vertices on $L(V)$ the *low degree vertices* and the vertices in $H(V)$ the *high degree vertices*. In any orthogonal drawing of G, there must be at least one zero degree angle at each vertex in $H(V)$.

The network $N = (V_N, E_N)$ with $E_N = E_{ff} \cup E_{vf} \cup E_{fv}$ is defined as follows:

$$\begin{aligned}
V_N &= V \cup F \\
E_{ff} &= \{(f_1, f_2) \mid f_1 \text{ and } f_2 \text{ are adjacent}\} \\
E_{vf} &= \{(v, f) \mid v \in L(V),\ v \text{ is on the boundary of } f\} \\
E_{fv} &= \{(f, v) \mid v \in H(V),\ v \text{ is on the boundary of } f\}
\end{aligned}$$

The cost per unit of flow and the capacities of the arcs are:

$$\begin{array}{rcl}
cap(e) = & \infty & \forall\, e \in E_{ff} \\
cap(e) = & 4 - deg(v) & \forall\, (v, f) \in E_{vf} \\
cap(e) = & 1 & \forall\, (f, v) \in E_{fv} \\
cost(e) = & 1 & \forall\, e \in E_{ff} \cup E_{fv} \\
cost(e) = & 0 & \forall\, e \in E_{vf}
\end{array}$$

The supply of each node is:

$$\begin{array}{rcl}
sup(v) = & 4 - deg(v) & \forall\, v \in V \\
sup(f) = & 4 - deg(f) & \forall\, f \in F - \{f_o\} \\
sup(f_o) = & -4 - deg(f_o) &
\end{array}$$

Note that the supply of a node in the network can be negative, and thus becomes a demand. The sum of all supplies is:

$$\sum_{f \in F - \{f_o\}} (4 - deg(f)) + \sum_{v \in V} (4 - deg(v)) - 4 - deg(f_o) \quad .$$

To calculate the value of this sum, we use the following facts:

1. A planar connected graph has exactly $|E| - |V| + 2$ faces.
2. Each edge is incident to exactly two vertices. Therefore, the sum of the degrees of all vertices is $2|E|$.
3. Each edge is on the boundary of exactly two faces in each embedding of a planar biconnected graph. Therefore, the sum of the degrees of all faces is $2|E|$.

Using these facts, we can easily verify that the sum of the supplies in N is exactly zero, so the supply is just as high as the demand.

This completes the definition of the Network N. A feasible flow in N describes an orthogonal representation and we will now show how to construct it from the flow values. Let $fl : E_N \to \mathbb{N}_0$ be a feasible flow in N. This means that it satisfies the following properties:

$$\forall \quad v \in V_N : \quad \sum_{e=(v,w)\in E_N} fl(e) - \sum_{e=(w,v)\in E_N} fl(e) = sup(v) \tag{4.6}$$

$$\forall \quad e \in E_N : \quad fl(e) \le cap(e) \tag{4.7}$$

$$\forall \quad e \in E_N : \quad fl(e) \ge 0 \tag{4.8}$$

Equality (4.6) states that the difference of the amount of flow that leaves a node and the amount of flow that enters the node is equal to the supply of the node. Inequality (4.7) states that the flow on each arc is at most as large as the capacity of that arc. The last inequality (4.8) guarantees that all flows are positive.

We will now show how a feasible flow fl in the network N can be translated to an orthogonal representation for G where the number of bends is equal to the cost of fl. We cannot directly compute the function b that assigns a bit string to each half edge (and thus the bends on each edge) from the flow fl. Here are the reasons:

1. Let $e = (f_1, f_2)$ be an arc in E_{ff}. Then the flow on this arc determines the bends on all the edges that separate f_1 and f_2. If there are several edges on the boundary separating the two faces, we can distribute the bends among these edges arbitrarily as long as the sequence of the bends is preserved.

2. The flow on arc $e = (f, v)$ in E_{fv} not only determines an angle between neighboring edges incident to v but also causes a bend if it is equal to one. So the bend string for each edge depends not only on the flow on the arcs in E_{ff} but also on the flow on the arcs in E_{fv}.

We will first show how to compute the function a that determines the angles between neighboring half edges incident to the same vertex by looking at the flow on the arcs in E_{vf} and E_{fv}. The correspondence between flows in N and an orthogonal representation of G is the same as described in (9). The flow on the arcs in E_{fv} additionally defines some of the bends. We will later incorporate these additional bends into the bend strings we compute using the flow on the arcs in E_{ff}.

- Let $e = (v, f)$ be an arc in E_{vf} and $\vec{e}_1 = (w, v) \in \vec{E}$ be the half edge incident to v that is on the boundary of f. Then we set

$$a(\vec{e}_1) = 90(fl(e) + 1) \quad .$$

- Let $e = (f, v)$ be an arc in E_{fv} and $\vec{e}_1 = (w, v) \in \vec{E}$ be the half edge incident to v with the property that f is to the left of $\vec{e}_1$. Let f' be the face with the reversal $r(\vec{e}_1)$ of $\vec{e}_1$

on its boundary and let $\vec{e}_2 = (x, v)$ be the predecessor of $r(\vec{e}_1)$ on the boundary of f'. Figure 4.15 shows this situation. Then we set

$$a(\vec{e}_2) = 90(1 - fl(e)) \quad .$$

If $fl(e) = 1$, we place one bend each on half edge $\vec{e}_1$ and half edge $r(\vec{e}_1)$. The bend on edge $\vec{e}_1$ is the last bend on the edge and is a bend to the left. Since $\vec{e}_1$ and $r(\vec{e}_1)$ are drawn as the same edge in a drawing of G, the bend on $r(\vec{e}_1)$ is the first bend on $r(\vec{e}_1)$ (remember that $r(\vec{e}_1)$ is directed towards w) an is a bend to the right.

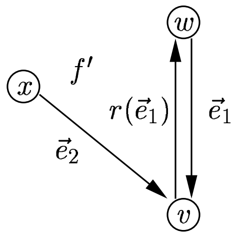


Figure 4.15: The half edges $\vec{e}_1$, $r(\vec{e}_1)$ and $\vec{e}_2$ in the situation where there is an arc from f to v in the flow network.

This defines the function a and the position and direction of the vertex bends. Now we want to define the function b that assigns bit-strings to each edge of G. These bit-strings determine the bends on the edges in a drawing that realizes the orthogonal representation.

We assume that the flow fl in network N is a flow with minimum cost. This implies the following fact that holds for all pairs of adjacent faces f_1 and f_2:

$$fl((f_1, f_2)) = 0 \ \vee \ fl((f_2, f_1)) = 0$$

The reason is that fl is a flow with minimum cost. If the flows on the arc (f_1, f_2) and (f_2, f_1) were both positive, we would be able to construct another feasible flow with smaller cost as follows: Assume w.l.o.g. $fl((f_1, f_2)) \leq fl((f_2, f_1))$. Then we can construct another feasible flow fl' from fl by setting $fl'((f_1, f_2)) = 0$ and $fl'((f_2, f_1)) = fl((f_2, f_1)) - fl((f_1, f_2))$. For all other arcs, the flow values of fl and fl' are the same. It is easy to verify that the feasibility of fl implies the feasibility of fl' and that the cost of fl' is smaller than the cost of fl by the amount $fl((f_1, f_2))$.

A flow of value $fl(e)$ on arc $e = (f_1, f_2) \in E_{ff}$ causes $fl(e)$ additional bends on the edges that separate f_1 and f_2 (additional to the bends caused by flow on the arcs in E_{fv}). Let B be the set of edges that separates f_1 and f_2. Then the flow causes $fl(e)$ bends to the right when we traverse the edges in B such that we leave f_1 to the right.

If the size of B is greater than one, it does not matter how we distribute the $fl(e)$ bends among the edges of B. So we can put them all on a single edge. Since we want the flow to define the function b in a unique way, we assume that all edges of G are numbered and that we place all bends on the edge of B with the lowest number. If we look at the flow on all the edges in E_{ff}, and set the additional bends according to the flow, then this defines the function b that assigns a bit string to each edge.

The function a and b together with the embedding Π define an orthogonal representation O of G. The reason is that the flow conservation constraints together with the supplies of the vertices in the network make sure that the functions a and b satisfy the constraints 4.2 to 4.5 on page 88.

The number of bends in the orthogonal representation O is equal to the cost of the flow because we create one bend for each unit of flow on arcs in E_{fv} and one bend for each unit of flow on arcs in E_{ff}. A proof for the fact that each minimum cost flow corresponds to an orthogonal representation with the minimum number of bends can be found in Bertolazzi, Battista, and Didimo (9).

4.3.5 Translation of the Flow Network into a Linear Program

Standard network flow problems can easily be translated into linear programs. In fact, the solution of a linear program representing a minimum cost flow network with integer capacities found by the simplex algorithm is always integer because the constraint matrix is totally unimodular (see for example Theorem 2.11 in (44)).

Let $N = (V_N, E_N)$ be the flow network from the previous section. We now present a linear program where each optimal solution represents a flow in N with minimum cost. So our objective function minimizes the flow on the arcs with cost one while the constraints guarantee that each solution represents a feasible flow.

We introduce one variable $f_e \in \mathbb{Q}$ for each arc e in E_N. In a feasible solution of the ILP, the variable f_e represents the flow on arc e (the value of the function fl for arc e).

LP 1

$$\min \sum_{e \in E_N} cost(e) \cdot f_e$$

subject to

$$\begin{array}{rcll}
\displaystyle\sum_{e=(v,w)\in E_N} f_e - \sum_{e=(w,v)\in E_N} f_e & = & 4 - deg(v) & \forall v \in V \\
\displaystyle\sum_{e=(f,w)\in E_N} f_e - \sum_{e=(w,f)\in E_N} f_e & = & 4 - deg(f) & \forall f \in F - \{f_o\} \\
\displaystyle\sum_{e=(f_o,w)\in E_N} f_e - \sum_{e=(w,f_o)\in E_N} f_e & = & -4 - deg(f_o) & \\
f_e & \leq & 4 - deg(v) & \forall e = (v,f) \in E_{vf} \\
f_e & \leq & 1 & \forall e = (f,v) \in E_{fv} \\
f_e & \geq & 0 & \forall e \in E_N
\end{array}$$

It is easy to see that the constraints of LP 1 are equivalent to the properties 4.6 to 4.8 on page 94 that describe a feasible flow. In the next sections, we want to combine this LP with the

ILP from Section 4.2 that describes all embeddings of a biconnected graph. This will enable us to construct a mixed integer linear program that describes all orthogonal representations of a biconnected graph.

4.4 The Integer Linear Program Describing All Orthogonal Representations of a Graph

In Section 4.2, we have introduced an ILP whose solutions are the combinatorial embeddings of a planar biconnected graph. In the previous section, we have introduced a linear program where an optimum solution corresponds to a simplified orthogonal representation with the minimum number of bends for a planar graph and a fixed embedding. In this section, we want to combine the ILP and the linear program. The result is a mixed integer linear program where an optimal solution corresponds to a simplified orthogonal representation of a planar biconnected graph with the minimum number of bends over all planar embeddings of the graph.

The first problem we have to tackle is that our ILP only describes the *combinatorial* embedding of a graph and not the *planar* embeddings. But this is easily solved by introducing new variables that determine the outer face of the embedding. This yields an ILP that describes all planar embeddings of a planar biconnected graph.

The more challenging problem is that we do not know which cycles in the graph will be face cycles in the solution. Thus, we have to introduce a new network similar to the network from Section 4.3.4 where we do not have to know during the construction which cycles are face cycles. When we have constructed the network, we will present a linear program that describes the feasible flows in the network.

And finally, we merge the ILP that describes the planar embeddings of a planar biconnected graph and the linear program that describes our new network and get an mixed integer linear program where the optimum solution corresponds to a simplified orthogonal representation with the minimum number of bends over all planar embeddings of a planar biconnected graph.

4.4.1 The Integer Linear Program that Describes All Planar Embeddings of a Biconnected Planar Graph

The ILP in Section 4.2 describes all combinatorial embeddings of a planar biconnected graph. If we want to determine an orthogonal representation, we need a planar embedding. Thus, we also need to fix the outer face of the embedding.

In a combinatorial embedding, any face can be chosen as the outer face to produce a planar embedding. Since the variables of the ILP of Section 4.2 correspond to the directed cycles in a graph that are face cycles in at least one combinatorial embedding, we can just introduce an additional binary variable for each variable in the original ILP. This binary variable is equal to one if the corresponding cycle is chosen as the outer face cycle and zero otherwise.

The new *outer face variables* must satisfy the following properties:

1. In every solution, exactly one outer face variable is equal to one and all others are equal to zero. This models the fact that in each planar embedding, exactly one face is the outer face.

2. An outer face variable can only be equal to one, if the corresponding face cycle variable of the original ILP is equal to one. So a directed cycle can only be chosen as the outer face cycle if the cycle is a face cycle.

3. Each outer face variable must either be equal to one or to zero.

So let o_c be the outer face variable for the cycle c. We assume that c belongs to the set C of all cycles of G that are a face cycle in at least one embedding of G. It follows that there is a cycle variable x_c for cycle c. Then the constraints we have to add to the ILP from Section 4.2 to describe planar embeddings are the following:

$$\begin{aligned} \sum_{c \in C} o_c &= 1 && (4.9) \\ x_c - o_c &\geq 0 \quad \forall\, c \in C && (4.10) \\ o_c &\in \{0,1\} \quad \forall\, c \in C && (4.11) \end{aligned}$$

If every variable o_c satisfies the constraints above, and the variables x_c satisfy all constraints in the ILP for combinatorial embeddings, then we can be sure that the solution indeed describes a planar embedding of the graph.

4.4.2 The Flow Network for Orthogonal Representation for Arbitrary Embeddings

Again we assume that C is the set of all cycles in G that are a face cycle in at least one embedding of G. When we build the network N', we do not know which of the cycles really are faces in the embedding that will determine the flow. Therefore, we have to introduce one vertex in N' for every cycle in C.

We define the network as $N' = (V_{N'}, E_{N'})$ with $V_{N'} = V \cup C$ (the set V is the set of vertices in G). The arc set consists of the three sets E_{vc}, E_{cv} and E_{cc}. These three sets are defined just like in Section 4.3.4. The only difference is that we deal with cycles instead of faces.

Again we define $L(V)$ as the set of vertices in V whose degree is at most four and $H(V)$ as the set of vertices with higher degree. For each half edge e, the edge $r(e)$ denotes the reversal edge of e. We say that two cycles are neighbors, if there is a half edge e in the first cycle with the property that $r(e)$ is contained in the second cycle. Then we can define the three arc sets as follows:

$$\begin{aligned} E_{cc} &= \{(c_1, c_2) \mid c_1 \text{ is a neighbor of } c_2\} \\ E_{vc} &= \{(v, c) \mid v \in L(V),\ v \text{ is on } c\} \\ E_{cv} &= \{(c, v) \mid v \in H(V),\ v \text{ is on } c\} \end{aligned}$$

We assume that the chosen embedding is defined by the cycle variables x_c and by the variables o_c. So there is one binary variable x_c for each cycle in C. The cycle is a face if and only if this variable is equal to one and o_c is equal to one if c is the outer face.

What we want to achieve is that the capacity of all the arcs in N' that start or end in a cycle c that is not a face cycle is zero. This is done by defining the capacity as x_c multiplied by an upper bound for a feasible flow on this arc. If c is not a face, the flow on the corresponding arcs must be zero. This has the same effect as taking c completely out of the network.

This is straight forward for the arcs in the sets E_{vc} and E_{cv}, because the corresponding arcs in the network of Section 4.3.4 have a fixed capacity. But there is no upper bound defined for the arcs that connect two faces in the original network.

We use the fact that the overall flow in the network is bounded by the supply of all nodes. It is not possible that the flow over an arc is greater than the total number of flow units produced in the network. So we take a look at the supplies defined in the network for a fixed embedding:

$$\begin{aligned} sup(v) = &\quad 4 - deg(v) & \forall\, v \in V \\ sup(f) = &\quad 4 - deg(f) & \forall\, f \in F - \{f_o\} \\ sup(f_o) = &\ -4 - deg(f_o) \end{aligned}$$

We want to compute the sum of all flow units produced in the network. Since G is biconnected, each vertex of G has at least degree two. So any node in N' corresponding to a vertex of G produces at most two units of flow. Since the graph G contains no self loops, every face in an embedding of G produces at most two units of flow. In every embedding of G, there are exactly $|E| - |V| + 1$ inner faces (the outer face only consumes units of flow). Thus, the total amount of flow produced in the network is at most

$$f_{max} = 2|V| + 2(|E| - |V| + 1) = 2|E| + 2 \quad .$$

It is not possible that a feasible flow assigns more than f_{max} units of flow to an arc. We can use this value as the maximum capacity of arcs in E_{cc} without excluding feasible flows. So we define the capacities of the arcs in N' as follows:

$$\begin{aligned} cap(e) = &\quad \min\{f_{max}x_{c_1}, f_{max}x_{c_2}\} & \forall\, e = (c_1, c_2) \in E_{ff} \\ cap(e) = &\quad x_c(4 - deg(v)) & \forall\, (v, c) \in E_{vc} \\ cap(e) = &\quad x_c & \forall\, (c, v) \in E_{cv} \end{aligned}$$

The supply of the nodes in N' that correspond to vertices in G is independent of the chosen embedding. The supply of nodes that correspond to cycles in G is zero if these cycles have not been chosen as face cycles. Otherwise, the supply depends on the value of the outer face variable for the cycle. We define $len(c)$ as the number of edges on the boundary of c (we do not use $deg(c)$ to make clear that c is a cycle and not a face). Then the supplies are defined as follows:

$$\begin{aligned} sup(v) = &\quad 4 - deg(v) & \forall\, v \in V \\ sup(c) = &\quad x_c(4 - len(c)) - 8o_c & \forall\, c \in C \end{aligned} \tag{4.12}$$

The interesting equality is Equality (4.12). It guarantees the following properties:

1. If the variable x_c is zero, the supply of c is zero. Note that the fact that x_c equals zero implies that o_c is zero.

2. If the variable x_c is one and the variable o_c is zero (c was chosen as an inner face), the supply of c is $4 - len(c)$.

3. If the variable x_c is one and the variable o_c is one (c was chosen as the outer face), the supply of c is $-4 - len(c)$.

The costs of the arcs arc defined in the same way as for network N:

$$\begin{aligned} cost(e) = &\quad 1 \quad \forall\, e \in E_{cc} \cup E_{fc} \\ cost(e) = &\quad 0 \quad \forall\, e \in E_{cf} \end{aligned}$$

The network is constructed in such a way that if the variables x_c and o_c of a planar embedding Π of G are given, the network behaves exactly like the network in Section 4.3.4 constructed for Π. The reason is that the nodes in the network that correspond to cycles that are not face cycles in the embedding have supply zero and all incident arcs have capacity zero. This has the same effect as taking them out of the network.

4.4.3 Formulation of the New Flow Network as a Linear Program

Just as the network N in Section 4.3.4 was translated into a linear program in Section 4.3.5, we will now transform the network N' from the previous section into a linear program. Just as in the last section, we assume that we are given binary values x_c and o_c for each cycle c in C that represent a planar embedding of G.

We introduce flow variables f_e for every arc $e \in E_{N'}$. The objective function of the linear program minimizes the cost of the flow. The constraints make sure that the difference between the amount of flow that leaves a node and the amount of flow that enters it is always equal to the supply of that node and that the flow on each arc does not exceed the capacity of the arc.

LP 2

$$\min \sum_{e \in E_N} cost(e) \cdot f_e$$

subject to

$$\sum_{e=(v,w) \in E_N} f_e - \sum_{e=(w,v) \in E_N} f_e = \quad 4 - deg(v) \qquad \forall v \in V$$

$$\begin{array}{rcl}
\sum_{e=(c,w)\in E_N} f_e - \sum_{e=(w,c)\in E_N} f_e = & x_c(4 - len(c)) - 8o_c & \forall c \in C \\
f_e \leq & x_c(4 - deg(v)) & \forall e = (v,c) \in E_{vc} \\
f_e \leq & x_c & \forall e = (c,v) \in E_{cv} \\
f_e \leq & x_{c_1} f_{max} & \forall e = (c_1,c_2) \in E_{cc} \\
f_e \leq & x_{c_2} f_{max} & \forall e = (c_1,c_2) \in E_{cc} \\
f_e \geq & 0 & \forall e \in E_N
\end{array}$$

The advantage of this linear program compared to the linear program for a fixed embedding is that we can compute a minimum cost flow for any planar embedding of G without rewriting the linear program. This enables us to combine the ILP for planar embeddings with the linear program we have just given to produce an mixed integer linear program where an optimum solution corresponds to a simplified orthogonal representation of the graph with the minimum number of bends.

4.4.4 The Complete Mixed Integer Linear Program

In this section, we describe the mixed integer linear program (MILP) for a biconnected planar graph where an optimal solution corresponds to a simplified orthogonal representation with the minimum number of bends. There are three types of variables in the MILP:

1. The cycle variables are binary. They determine for each cycle in the set C of potential face cycles if they are indeed a face cycle in the simplified orthogonal representation described by the solution.

2. The outer face variables are also binary. They determine if a cycle is the outer face cycle in the simplified orthogonal representation.

3. The flow variables are continuous. They determine the flow on each arc in the network N' from Section 4.4.2. These flow values determine the angles between edges incident to the same vertex in the orthogonal representation and the number and directions of the bends on each edge.

The objective function is the same as in the linear program that describes the flow network for arbitrary embeddings in the previous section. So we minimize the units of flow on the arcs that connect cycles of the graph and on the arcs that start in cycles and end in vertices. The flow on these arcs is equal to the number of bends in the simplified orthogonal representation that corresponds to the flow. Since there are only flow variables in the objective function, the values of the binary variables have only indirect influence on the objective function value via the constraints that connect them to the flow variables.

The constraint matrix can be separated into three parts. The first part contains all the constraints in the ILP from Section 4.2 that describes combinatorial embeddings. These constraints contain only the cycle variables.

The second part of the constraint matrix consists of the additional constraints that were introduced in Section 4.4.1 to describe planar embeddings. These constraints contain cycle variables and outer face variables and guarantee that exactly one of the cycles chosen as a face of the embedding is chosen as the outer face.

The third and last part consists of all the constraints introduced in the last section. These constraints contain all three types of variables and they are the only constraints that contain the continuous flow variables. They guarantee that the flow computed in the optimization process is consistent with the planar embedding described by the cycle variables and outer face variables and that the flow describes the bends and angles of an orthogonal representation.

4.5 The Algorithm for Minimizing the Number of Bends

In this section, we describe how we actually compute orthogonal representations of graphs with the minimum number of bends. In Section 4.5.1 we present a high level overview of our algorithm. We compute a first MILP, and optimize it with a mixed integer solver. Then we try to separate the constraints that we do not put into the initial MILP because of their exponential number. If there are no more violated constraints, the optimization is finished and we compute the orthogonal representation that corresponds to the solution. Otherwise, we throw in the violated constraint and re-optimize.

In Section 4.5.3, we will describe some techniques we use to speed up the computation. We will show that we can get rid of some variables and constraints by exploiting certain properties of planar embeddings.

4.5.1 High Level Structure of the Algorithm

An important factor in our algorithm is that we cannot compute the complete MILP before we optimize. The reason is that there is a potentially exponential number of subtour elimination constraints that have to be satisfied by any feasible solution for the ILP that describes combinatorial embeddings.

We solve this problem as already described in Section 4.2.4 on page 74: During the recursive construction of the embedding ILP we compute for each P-node a data structure that enables us to check efficiently, if a solution satisfies the subtour elimination constraints for that P-node. This data structure consists of a complete graph and a mapping from the cycle variables in the ILP to the edges in this graph. The graph is the instance of the asymmetric traveling salesman problem (ATSP) that is equivalent to the embedding problem of the P-node skeleton.

When we have found a solution, we look at all the data structures constructed for all P-nodes and set the weights of the edges in the ATSP according to the values of the corresponding cycle variables. Then we use a minimum cut algorithm to check if there are any violated subtour elimination constraints. If this is the case, we can easily construct the subtour elimination constraint by looking at the edges contained in the minimum cut and the corresponding variables of the MILP. Then we add the new inequality to the MILP and re-optimize.

```
Input: A biconnected planar graph $G$
Output: A simplified orthogonal representation of $G$ with the minimum number of
        bends
Compute the embedding constraints CON 1;
Compute the constraints CON 2 for the outer face variables;
Compute the constraints CON 3 for the flow network;
Build the MILP from CON 1, CON 2, and CON 3;
repeat
    Compute an optimal solution for the MILP;
    if Solution violates any subtour elimination constraints;
    then
        Add the violated subtour elimination constraint to the MILP;
        Mark the solution as infeasible;
    else
        Mark the solution as feasible
until Solution is marked feasible;
Transform the solution into a simplified orthogonal representation;
```

To compute the constraints in `CON 1`, we first construct the SPQR-tree of the graph, then we choose a split node and recursively compute the ILPs for the resulting split trees, as described in Section 4.2.6 on page 79. Then the ILPs are merged into a single ILP that describes the combinatorial embeddings of the graph.

After `CON 1` is computed, we can easily compute the ILP for the outer face variables since the set of cycles of the graph that are face cycles in at least one embedding of the graph has been computed during the construction of `CON 1`. The outer face variables and the cycle variables are then incorporated into the linear program that describes a simplified orthogonal representation (the constraints of `CON 3`).

Together with the objective function that minimizes the number of bends in the orthogonal representation, the constraints of `CON 1`, `CON 2` and `CON 3` are then used as input for the mixed integer solver of CPLEX (34). The result is a solution where all variables are integer. This is true because the capacities in the flow network described by the constraints in `CON 3` are integer and a linear program describing a min cost flow problem with integer capacities produces an integer flow.

The solution produced in this way may violate subtour elimination constraints computed for a P-node in the SPQR-tree for the graph. Therefore, we check the solution by testing for each P-node skeleton, if one of the subtour elimination constraints of the skeleton is violated. When we construct the ILP that describes the combinatorial embeddings of a graph, we also compute a special graph for each P-node skeleton that enables us to check if a subtour elimination constraint for the skeleton is violated by a solution (see Section 4.2.4 on page 74). If we find a violated subtour elimination constraint, we add it to the ILP and re-optimize. Otherwise, we have found an optimal feasible solution.

In the final step, we have to transform the solution of the ILP into a simplified orthogonal representation. Each feasible solution describes a flow in a network. If we delete from this network all the vertices that correspond to cycles that are not face cycle in the solution, we get the same network as in (9). Therefore, we can use the same approach used in (9) to translate the flow in the network to an orthogonal representation for the problem graph (see Section 4.3.4 on page 92 for the details of the transformation of the flow to an orthogonal representation).

4.5.2 Modifications to Improve Performance

In our implementation, we use several methods to speed up the computation. They introduce additional complexity into the algorithm but lead to significant improvements in performance. In this section, we explain why they are useful and how they work.

In our computational experiments with benchmark graphs, we observed that it is quite rare that the SPQR-tree of a graph has a P-node skeleton with more than four edges. Therefore, it makes sense to speed up the computation of the ILP for the special cases of P-node skeletons with three and four edges.

If we consider a P-node skeleton with three edges, it is not hard to see that it has only two different combinatorial embeddings that are mirror images of each other. So we can treat these skeletons in exactly the same way as we treat the triconnected skeletons of R-nodes. In Section 4.2.3 on page 73, we have given the complete description of the polytope of combinatorial embeddings of a graph with only two embeddings that we use for R-nodes. In our implementation, we use the same description for the combinatorial embeddings of a P-node skeleton with three edges.

For P-node skeletons with four edges, we used a different approach. Such a skeleton has six different combinatorial embeddings. So we used the PORTA system (17) to compute a complete description of the polytope defined by the six embeddings of the skeleton of a P-node skeleton with four edges. The constraints are contained in the code of our implementation and so we do not have to compute them at run time. Another advantage is that we do not have to separate any subtour elimination constraints if the SPQR-tree for a graph does not contain a P-node whose skeleton has more than four edges.

Another fact that we use to speed up the computation of the solution is that we do not need an outer face variable for each cycle variable. The reason is that for each directed cycle c that is represented by a variable in the ILP, there is another variable in the ILP that represents the directed cycle $\bar{c}$ that we get by reversing all edges in c.

Let P be the set of all planar embeddings of the graph G where c is chosen as the boundary of the outer face. Let $\bar{P}$ be the set of planar embeddings of G where $\bar{c}$ is chosen as the boundary of the outer face. There exists a bijection $b : P \to \bar{P}$ that maps each planar embedding in P to the planar embedding in $\bar{P}$ that we get by reversing all face cycles. So the embedding $b(\Pi)$ is the mirror image of the embedding Π. It follows that the number of bends in a bend minimum orthogonal drawing that realizes Π is the same as in a drawing of $b(\Pi)$ because the two drawings are mirror images of each other.

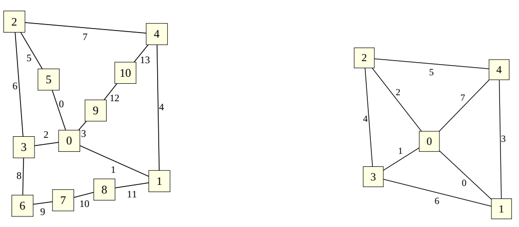
We are interested in finding a single bend minimal embedding of G. As long as we can guarantee that the solution computed is the best among all possible solutions, it is not important that we only optimize over a subset of all possible solutions. Therefore, we only introduce outer face variables for half the cycles that are represented by cycle variables. We make sure that there is exactly one outer face variable for each *undirected* cycle that is represented by a cycle variable.

This makes the computation of a feasible solution easier, because we get rid of a quarter of all integer variables. Without this optimization, we have one cycle variable and one outer face variable for each directed cycle that is a face cycle in at least one embedding. Since we got rid of half of the outer face variables, we saved a quarter of the integer variables (remember that the flow variables are not declared as integer variables in the MILP because the integrality of the cycle variables and the outer face variables together with the integer capacities of the arcs in the flow network force the flow variables to be integer).

4.5.3 Preprocessing

In the preprocessing step, the original problem is transformed into a different problem of smaller size. After this problem is solved, we transform the solution for the smaller problem into a solution for the original problem.

This is done as follows: Assume that the edges $e_1, \ldots e_k$ form a path of maximal length in G with the property that all inner vertices of the path have degree two. Paths with this property are called *primitive paths*. We replace this path with a single edge connecting the vertex at the start of the path with the vertex at the end of the path. This operation is called *collapsing the path*. Collapsing all primitive paths results in the *reduced graph* G' of G. Figure 4.16 shows a graph and its reduced graph.



(a) (b)

Figure 4.16: A graph (left) and its reduced graph (right).

Note that collapsing a path in a biconnected graph only produces a self loop (an edge that starts and ends at the same vertex), if the whole graph is a simple cycle. Producing an orthogonal representation with the minimum number of bends for this case is easy. If the graph has more than three nodes, we can easily produce an orthogonal representation without bends by choosing four arbitrary vertices as the corners of the rectangle that will form the shape of the drawing of the graph. If there are three vertices in G, we have to put one bend on one of the edges to produce a rectangle. So in the rest of this section, we assume that G is not a simple cycle. Thus, the reduced graph of G does not contain self loops.

Collapsing all primitive paths does not alter the number of embeddings. The reason is that embeddings can be defined by the circular sequence of the edges around each vertex. The circular sequence of the edges around a vertex with degree two is always the same. All vertices with degree greater than two still have the same degree after the collapsing step.

When we look at the SPQR-tree, collapsing a primitive path causes the set of Q-nodes corresponding to the edges on the primitive path to merge into a single Q-node. All these Q-nodes are incident to the same S-node. This can be seen as follows: Let P be the set of edges on the primitive path. If we choose any of these edges as the reference edge for constructing the SPQR-tree, then the root of the Proto-SPQR-tree must be an S-node because at least one of the end nodes of the path is a cut vertex of the graph when we delete the reference edge. All edges in P produce a Q-node attached to the S-node. As we have seen in Section 2.4, the choice of the reference edge is irrelevant for the structure of the SPQR-tree. So all the Q-nodes representing the edges of P are adjacent to the same S-node.

When the Q-nodes merge, the adjacent S-node vanishes if its degree after the merger is only two. In this case, the Q-node produced by the merger is directly attached to the second node adjacent to the S-node (note that this second node is not a Q-node because the graph is not a simple cycle). If the degree of the S-node after the merger is greater than two, the S-node will not vanish. In any case, the number of R- and P-nodes in the tree does not change and thus the number of embeddings stays the same.

Figure 4.17 shows the SPQR-trees for the graphs in Figure 4.16. The numbers of the Q-nodes correspond to the numbers of the edges in the graphs. Replacing the primitive paths with single edges resulted in the removal of all S-nodes because the children of each S-node were merged into a single Q-node.

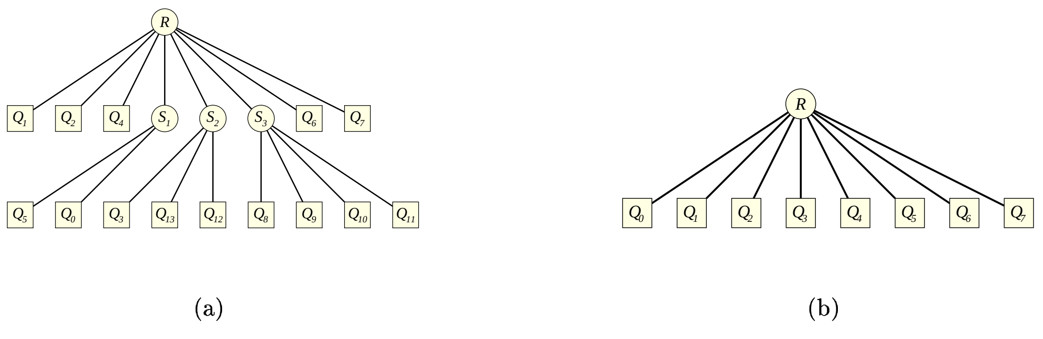


Figure 4.17: The SPQR-trees of the graphs shown in Figure 4.16.

Let P be the set of planar embeddings of G and P' the set of planar embeddings of G'. Every embedding in P' can be constructed from the corresponding embedding in P by collapsing all primitive paths. On the other hand, every embedding in P can be obtained from the corresponding embedding in P' by replacing all edges corresponding to primitive paths in G by the original paths.

So the translations of embeddings between the original graph and the reduced graph is not difficult. Assume that we have produced an orthogonal drawing of the reduced graph. We can transform this drawing into a drawing of the original graph by drawing the inner vertices of the primitive paths on the drawings of the edges that replaced the primitive paths. The problem is that an orthogonal representation with the minimum number of bends for the reduced graph does not always define an orthogonal representation for the original graph with the minimum number of bends.

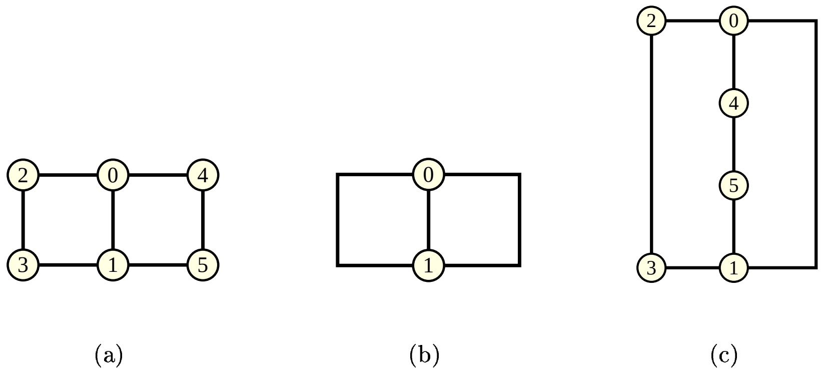


Figure 4.18: Reducing the graph 4.18(a) results in graph 4.18(b). Orthogonal representation 4.18(c) has been obtained by translating a bend minimal orthogonal representation of the reduced graph into an orthogonal representation for the original graph.

Consider the graph G in Figure 4.18(a). Its reduced graph G' is shown in Figure 4.18(b). If we only consider the problem of finding an orthogonal representation of G', it is not hard to see that we need at least four bends. We can pick two arbitrary edges of the graph and place two bends on each of them to produce an orthogonal representation that looks like the one represented in Figure 4.18(b).

Nothing keeps us from putting two of these bends on the edge that represents the edge $(0, 1)$ in G. A drawing that represent a corresponding orthogonal representation of G is shown in Figure 4.18(c). This is clearly not an orthogonal representation of G with the minimum number of bends since it has two bends while it is obviously possible to draw G with no bends at all (as shown in Figure 4.18(a)).

The problem is that reducing a graph destroys information that is needed in the optimization process. An edge in the reduced graph that represents a primitive path in the original graph should not be treated like an edge that represents an edge in the original path. The reason is that an edge that represents a path can take a certain number of bends without producing bends in the corresponding orthogonal representation of the original graph.

Assume that the orthogonal representation O' of the reduced graph G' has k bends on edge e. This edge represents a primitive path p with l inner vertices in the original graph G. In the translation of O' into an orthogonal representation O for G, we will place $\min\{k, l\}$ inner vertices of p on the bends of e to save bends.

When computing the reduced graph of G, we have to remember the number of inner vertices on every primitive path we replace with a single edge. When we minimize the number of bends for the reduced graph, we can use this information to place bends on edges that correspond to primitive paths in the original path. This enables us to save bends where possible.

The idea is the following: If an edge in the reduced graph represents a primitive path in the original graph with l inner vertices, then we allow l bends on this edge that have no cost. In the network from Section 4.4.2 on page 98, the flow between two nodes that represent faces is the number of bends on the boundary between the two faces. This is modeled by the costs of the arcs that connect the two faces in the network. If some edges on the boundary between the two cycles represent primitive paths in the original path, some units of flow should not produce costs, because we can hide the corresponding bends by placing vertices on them. Therefore, we introduce new arcs between the faces that have cost zero and whose capacity is equal to the number of bends that can be hidden by placing vertices on them.

There is a rather elegant way to model this complication in the flow network from Section 4.4.2. Let c_1 and c_2 be two cycles in a planar biconnected graph. We say that the edge e in G' *separates* c_1 and c_2, if one half edge of e belongs to c_1 and the other to c_2. The set $Sep(c_1, c_2)$ is the set of all edges that separate c_1 and c_2. In the network from Section 4.4.2, there are two arcs (c_1, c_2) and (c_2, c_1), if $Sep(c_1, c_2)$ is not empty.

Let c_1 and c_2 be directed cycles in the reduced graph G' of G that are face cycles in at least one embedding of G' and where the set $Sep(c_1, c_2)$ contains at least one edge that represents a primitive path in G. Then we insert an additional pair of arcs into the network. These arcs are $z_1 = (c_1, c_2)$ and $z_2 = (c_2, c_1)$. The costs of both arcs are zero. We call these arcs *zero cost arcs*.

If the two cycles c_1 and c_2 are chosen as face cycles of the embedding, the capacity of the arcs z_1 and z_2 is the sum z of the number of all inner vertices of primitive paths represented by edges in $Sep(c_1, c_2)$. This number z is the number of bends we can save by placing the inner vertices of the primitive paths on the bends. If at least one of the cycles c_1 and c_2 is not a face cycle, the capacity of the arcs is zero.

We modify the linear program on page 100 using the same approach by introducing two new variables f_{z_1} and f_{z_2} that model the flow on the zero cost arcs. These variables are included in the flow conservation constraints for the nodes in the network that correspond to cycles in the graph (Constraint (4.13) in the LP on page 100 of Section 4.4.3). Additionally, we need four constraints that set the capacity of the arcs z_1 and z_2. These constraints are as follows:

$$\begin{aligned}
f_{z_1} &\le x_{c_1} z \\
f_{z_1} &\le x_{c_2} z \\
f_{z_2} &\le x_{c_1} z \\
f_{z_2} &\le x_{c_2} z
\end{aligned}$$

These constraints guarantee that the flow on the zero cost arcs is zero if at least one of the cycles that they connect is not chosen as a face cycle and that the flow is at most the number of inner vertices on primitive paths on the boundary between the two cycles. Note that the new variables do not appear in the objective function, since flow on the corresponding arcs in the network does not produce cost.

To compute an orthogonal representation of the original graph G with the minimum number of bends, we first compute the SPQR-tree for the reduced graph G' of G. Then we use this SPQR-tree to compute the ILP that defines all embeddings of G'. The next step is to compute the flow network for G' where we insert the additional zero cost arcs for the edges in G' that represent primitive paths in G. This flow network defines a linear program which is added to the ILP that describes the planar embeddings of G' to produce the MILP.

Now we have to address the problem of constructing an orthogonal representation of G from the solution found for the MILP (we assume that the solution does not violate any subtour elimination constraints). Let K be the value of the objective function. We first compute an orthogonal representation for G' with $K' \geq K$ bends and then use this orthogonal representation for the reduced graph to compute an orthogonal representation for the original graph G with K bends.

In Section 4.3.4, we have shown how to translate a minimum cost flow into an orthogonal representation of the graph. Because we have introduced the zero cost arcs into the network, we have to take more care when we assign the bends to the edges of the reduced graph. Let c_1 and c_2 be two cycles in G' and B the set of separating edges of the two cycles. If there are no zero cost arcs connecting c_1 and c_2, we can proceed in the same way as in Section 4.3.4 to assign bends to the edges in B.

Now assume that there are zero cost arcs connecting c_1 and c_2. It is clear that we cannot have flow on the arcs with non-zero cost connecting c_1 and c_2 if there is no flow on the zero cost arcs. This would be a contradiction to the minimum cost of the flow we have computed. So if there is any flow between c_1 and c_2, there is flow on the zero cost arcs.

We can assume that in any solution we find, the flow on at least one of the zero cost arcs connecting c_1 and c_2 is zero. We can transform any solution where this is not the case easily into another feasible solution with the same cost where the flow on one of the zero cost arcs is zero. Assume that the flow from c_1 to c_2 on the zero cost arc is z_1 and the flow from c_2 to c_1 is z_2 with $z_1 \leq z_2$. Then we can set the flow from c_1 to c_2 to zero and the flow from c_2 to c_1 to $z_2 - z_1$. The solution obtained in this way is feasible. The maximum capacities are not exceeded because we diminished the flow on the two zero cost arcs. The flow conservation constraints are still satisfied because the flow entering and leaving c_1 and c_2 via the zero cost arcs is still the same. Since the flow on zero cost arcs does not influence the value of the objective function, the new solution has the same objective function value as the original solution.

So we assume w.l.o.g. that the flow from c_1 to c_2 on the zero cost arc is z with $z > 0$ while the flow on the zero cost arc from c_2 to c_1 is zero. This implies that the flow on the non zero cost arc from c_2 to c_1 is also zero. Otherwise, we could easily construct a feasible flow with smaller cost. We assume that the flow on the non zero cost arc from c_1 to c_2 is k with $k \geq 0$.

For each edge e on B, we define $i(e)$ as the number of inner vertices on the primitive path in G represented by e. If e does not represent a primitive path in G, $i(e)$ is zero. We use the following simple algorithm to assign bends produced by flow on zero cost arcs to the half edges of the edges in B:

```
$n = z$;
**for** $e \in B$ **do**
    $k' = \min\{n, i(e)\}$;
    $n = n - k'$;
    Assign $k'$ bends to the two half edges of $e$;
```

If all edges of the graph are numbered and the loop in the algorithm runs through the edges of B in the corresponding sequence, the orthogonal representation O' computed in this way is uniquely defined. Note that the number of bends K' in O' is greater than the objective value K of the MILP if flow variables for zero cost arcs are greater than zero. The reason is that the flow on the zero cost arcs corresponds to bends that we can hide under vertices of the original graph. These vertices are inner vertices of primitive paths and are therefore not contained in G'.

As we have already seen in Section 4.3.4, not all bends on the edges in the orthogonal representation are determined by the flow between nodes in the network corresponding to cycles. Some bends, called *vertex bends*, are determined by the flow on the arcs in E_{cv}. These are arcs that start in a node of the network corresponding to a cycle of the original graph and end in a node of the network corresponding to a vertex of the graph. The flow on these arcs corresponds to zero degree angles between neighboring half edges incident to the vertex and causes a bend on an edge incident to the vertex (a vertex bend).

One important observation is that we can't hide vertex bends under vertices. The reason is that the distance between edges that run parallel to each other because of zero degree angles may be arbitrarily small (the distance shrinks with the number of parallel edges). Therefore, the distance between a vertex bend and the next edge in the bundle of parallel edges can be arbitrarily small. So there is no space for the drawing of a vertex at a vertex bend (see Figure 4.19 for an example of vertex bends). To account for this fact, we set the vertex bends in the orthogonal representation O' of the reduced graph G' just like we would set them if G' was the original graph.

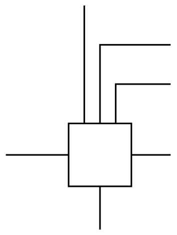


Figure 4.19: A vertex with two vertex bends in a podevsnef drawing.

When we transform the orthogonal representation O' computed for the reduced graph G' of G into an orthogonal representation O for G, we have to pay attention that we do not put a vertex of G that is not contained in G' (a degree two vertex) on a vertex bend. To avoid this, we have to mark these bends to distinguish them from regular bends in the orthogonal representation O'. We define two functions sb and tb on the half edges of G' in the following way:

$$\begin{aligned} sb : \vec{E}' \to \{0,1\}, \quad sb(\vec{e}) = 1 \quad &\Leftrightarrow \text{the first bend on } \vec{e} \text{ is a vertex bend} \\ tb : \vec{E}' \to \{0,1\}, \quad tb(\vec{e}) = 1 \quad &\Leftrightarrow \text{the last bend on } \vec{e} \text{ is a vertex bend} \end{aligned}$$

So we have computed an orthogonal representation O' for the reduced graph G' together with the functions sb and tb. We have to transform this orthogonal representation into a corresponding orthogonal representation of G with K bends. Transferring the embedding of O' to G is easy, because we only have to copy the circular sequence of the incident edges from the nodes of G' to the nodes of G. The circular sequence of the incident edges for the vertices of G that are not contained in G' does not matter because there are only two edges incident to these vertices.

After the transfer of the embedding, we transfer the angles between incident edges for each vertex of G' to the corresponding vertex of G. Remember that an orthogonal representation defines for each half edge $\vec{e}$ a value $a(\vec{e}) \in \{0, 90, 180, 270, 360\}$, which is the angle between e and the successor of e on the same face cycle.

First, we have to build a mapping em from the half edges of G' to the half edges of G as follows:

1. If e is contained in G (e does not represent a primitive path in G), we define $em(\vec{e})$ as the same half edge in G.

2. If e is not contained in G, there is a primitive path p in G that is represented by e. We choose $em(\vec{e})$ as the half edge on this path whose target is the same vertex as the target vertex of $\vec{e}$.

This defines a mapping from all half edges of G' to the half edges of G. Note that $em^{-1}(\vec{e})$ where $\vec{e}$ is a half edge in G is only defined for half edges whose target vertex is not an inner vertex of a primitive path of G.

So let a' be the function of O' that assigns angles to the half edges of G'. We first define the function a of O for the half edges $\vec{e}$ of G whose target is not a vertex with degree two (an inner vertex of a primitive path, replaced by a single edge in G'). So let $\vec{e}$ be a half edge in G with this property. Then we define:

$$a(\vec{e}) = a'(em^{-1}(\vec{e})) \tag{4.13}$$

Now we have defined the function a for all half edges of G whose target is not the inner vertex of a primitive path. To define the function a for the rest of the half edges, we need to know about the bends on the edges in G' that correspond to primitive paths in G. The next step is to look at the function b' of O' (the function that assigns bend strings to the half edges of G') and to translate each bend either into an angle of a half edge on a primitive path of G or into a bend on a half edge of G.

We go through all half edges $\vec{e}$ of the reduced graph G'. If $\vec{e}$ is also contained in G, the string $b(\vec{e})$ is the same as the string $b'(\vec{e})$. If $\vec{e}$ is not contained in G', it corresponds to one of the two paths $\vec{p}_1$ and $\vec{p}_2$ of half edges that belong to a primitive path P in G. The path $\vec{p}_2$ is called the *opposite path* of $\vec{p}_1$ and vice versa. We have not yet defined the function a for the half edges on $\vec{p}_1$ and $\vec{p}_2$ whose target vertex is an inner vertex of P (and thus has degree two).

We have to transform the bends defined by $b(\vec{e})$ into angles and bends for the half edges on $\vec{p}_1$ and $\vec{p}_2$. This means we have to define the function b for all the half edge on $\vec{p}_1$ and $\vec{p}_2$ and the function a for the half edges of the two paths whose target vertex has degree two (is an inner vertex of the primitive path). Doing this for all half edges representing primitive paths in G completes the definition of the functions a and b and thus of the orthogonal representation O of G.

We exploit the following fact: If we have determined the functions a and b for the half edges of $\vec{p}_1$, then the corresponding values for the half edges of $\vec{p}_2$ is easy to compute. Let the two paths be defined by the following sequences of half edges:

$$\begin{aligned}\vec{p}_1 &= \vec{e}_{1.1}, \vec{e}_{1.2}, \dots, \vec{e}_{1.l}\\ \vec{p}_2 &= \vec{e}_{2.1}, \vec{e}_{2.2}, \dots, \vec{e}_{2.l}\end{aligned}$$

We assume that the target vertex of $\vec{e}_{i.j}$ is the source vertex of $\vec{e}_{i.j+1}$ for $i \in \{1, 2\}$ and for $1 \leq j \leq l-1$. Then the two half edges $\vec{e}_{1.i}$ and $\vec{e}_{2.l+1-i}$ belong to the same edge e of G. See Figure 4.20 for an example with $l = 4$.

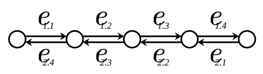


Figure 4.20: The two paths $\vec{p}_1$ and $\vec{p}_2$ of half edges for the case $l = 4$.

We assume now that we have already defined the functions b and a for all the half edges of $\vec{p}_1$. Then the definition of the function a for the half edges of $\vec{p}_2$ is as follows:

$$a(\vec{e}_{2.l-i}) = 360 - a(\vec{e}_{1.i}) \quad \text{for } 1 \leq i \leq l-1$$

Note that $\vec{e}_{2.l-i}$ is not the half edge that belongs to the same edge e of G as $\vec{e}_{1.i}$ but the half edge on p_2 that has the same target vertex as the half edge $\vec{e}_{1.i}$. Note also that the function a is only set for the half edges whose target vertex is not an inner vertex of the primitive path. The reason we set the angle to 360 minus the angle of the corresponding angle on $\vec{p}_1$ is that the vertices on the path have degree two. Therefore, there are exactly two angles at each vertex and we guarantee that the sum of the angles at each vertex is 360.

To set the function b for the half edges on $\vec{p}_2$, we use again the notation introduced in Section 4.3.2. Let $s = (s_1, \dots, s_k) \in \{0, 1\}^*$ be a string of zeroes and ones. We define $\bar{s}$ as follows: $\bar{s} = ((1 - s_k), \dots, (1 - s_1))$. So we get the string $\bar{s}$ by replacing the zeroes with ones and vice versa and then reversing the sequence of the elements. Now we can define the function b for the edges on $\vec{p}_2$:

$$b(\vec{e}_{2.l+1-i}) = \overline{b(\vec{e}_{1.i})} \quad \text{for } 1 \le i \le l$$

This defines the functions a and b for the path $\vec{p}_2$ of half edges if the functions a and b for the opposite path are already defined. Now we describe how to define the functions a and b for a path of half edges of a primitive path if the functions for the opposite path have not yet been computed. Our algorithm must meet the following requirements:

- The shape of the path $\vec{p}_1$ must be the same as the shape of the corresponding half edge $\vec{e}$ in G'.
- As many bends as possible on $\vec{e}$ should be realized by angles at vertices on $\vec{p}_1$ rather than by bends on half edges of $\vec{p}_1$. This is meant by placing a vertex on a bend or hiding a bend under a vertex.
- We cannot hide a vertex bend under a vertex.

Algorithm 4.5.3 computes the functions a and b for the edges on $\vec{p}_1$.

```
$s = b'(\vec{e})$;
**if** $sb(\vec{e}) = 1$ **then**
    Let $z$ be the first letter of $s$;
    Delete $z$ from $s$;
    Set the first letter of $b(\vec{e}_{1.1})$ to $z$;
**if** $tb(\vec{e}) = 1$ **then**
    Let $z$ be the last letter of $s$;
    Delete $z$ from $s$;
    Set the last letter of $b(\vec{e}_{1.l})$ to $z$;
**for** $i = 1$ *to* $l - 1$ **do**
    Set $b(\vec{e}_{1.i}) = ()$;
    **if** $s$ *is not empty* **then**
        Let $z$ be the first letter in $s$;
        Delete $z$ from $s$;
        **if** $z =$ '0' **then**
            Set $a(\vec{e}_{1.i}) = 90$;
        **else**
            Set $a(\vec{e}_{1.i}) = 270$;
    **else**
        Set $a(\vec{e}_{1.i}) = 180$;
Insert $s$ at the beginning of $b(\vec{e}_{1.l})$;
```

First, the algorithm makes sure that the vertex bends on half edge $\vec{e}$ are translated into real bends on the path $\vec{p}_1$. If there is a vertex bend at the start of $\vec{e}$, it is inserted as the first bend on the first half edge of $\vec{p}_1$. If there is a vertex bend at the end of $\vec{e}$, it is inserted as the last bend on the last half edge of $\vec{p}_1$.

Then we go through all the half edges of $\vec{p}_1$ except the last half edge. We set the bend string of each half edge to the empty string because we want to hide as many bends as possible under vertices. The angle at the target of the current half edge is determined by the first remaining bend on $\vec{e}$. If there is no bend left on $\vec{e}$, the angle is 180 degrees.

After the loop, we place all bends of $\vec{e}$ that have not yet been distributed on the last half edge of $\vec{p}_1$. Note that we do not set the angle of the last half edge because the target of this edge is not the inner vertex of a simple path. Therefore, the angle at this vertex has already been set using rule 4.13 on page 111.

This defines how we can translate the orthogonal representation O' of the reduced graph G' into an orthogonal representation O of G with the property that the number of bends in O is the objective value of the solution of our MILP. Remember that the number of bends in O' may be greater than this value because we can hide bends of O' that are caused by flow on zero cost arcs in the network under inner vertices of simple paths in G.

The algorithm above guarantees that as many bends as possible are hidden under vertices on simple paths. The number of the inner vertices on the boundaries between two faces is modeled by the zero cost arcs in the flow network. This ensures that the value of the minimum cost flow computed by solving the MILP is the number of bends in the orthogonal representation O.

4.6 Computational Results

We wanted to test the performance of our algorithms and compare it with the performance of the branch & bound algorithm developed by Bertolazzi, Battista, and Didimo (9). We also compared the results of these exact algorithms with the results obtained using a heuristic. Section 4.6.1 contains the descriptions of the algorithms we tested.

We used two different sets of biconnected planar graphs as our benchmark. Note that the branch & bound algorithm, like the algorithms developed in this chapter, is limited to planar biconnected graphs. Section 4.6.2 describes the origins of the two benchmark sets we used. We call the first set the *real world set*, and the second the *artificial set*.

In Section 4.6.3, we present the results for the real world benchmark set. Our data shows that our new approach performs better than the previously known branch & bound approach for the larger graph of the set. Another result is that the exact algorithms produce significantly better solutions than the heuristic for almost half of the graphs.

Section 4.6.4 contains the results for the artificial benchmark set. This is the set used by Bertolazzi, Battista, and Didimo (9) to test the performance of their branch & bound algorithm. On this benchmark set, our algorithm is about twice as fast as their algorithm.

We take a closer look at two graphs of the real world benchmark set in Section 4.6.5. One of them is among the smallest of the real world set while the other is among the largest. We present drawings of the graph computed by the algorithms we tested and present some details about the mixed integer linear programs produced by our algorithm.

4.6.1 The Algorithms We Tested

Using the benchmark graphs, we tested four different algorithms for computing orthogonal representations:

1. **B&B**: This algorithm finds an orthogonal representation with the minimum number of bends among all possible embeddings of the graph using a branch & bound approach (9). We used the implementation in the GDToolkit library (see `http://www.dia.uniroma3.it/~gdt/`).

2. **Mix**: This algorithm uses our formulation of the set of all orthogonal representations of a graph as a mixed integer linear program. The CPLEX mixed integer solver (34) is used to find an optimum solution. If necessary, subtour elimination constraints are separated.

3. **MixR**: The difference to the **Mix** algorithm is that the mixed integer program is not computed for the original graph but for the reduced graph. The result obtained for the reduced graph is then transformed into a result for the original graph as described in Section 4.5.3 on page 105.

4. **Heuristic**: This heuristic computes an arbitrary embedding of the graph and then constructs the flow network from Section 4.3.4 on page 92 that describes all orthogonal representations of a graph for a fixed embedding. A minimum cost flow algorithm is used to find an orthogonal representation with the minimum number of bends for the chosen embedding. We used the implementation in the GDToolkit library (see `http://www.dia.uniroma3.it/~gdt`) in our experiments.

4.6.2 The Two Benchmark Sets

The three exact algorithms we tested can only deal with biconnected planar graphs. Most graphs that come from commercial applications of graph drawing do not have these properties. However, we wanted to find out how the algorithms perform when used on real world graphs that have been modified to meet our requirements.

A method often used for drawing non-planar graph is the planarization method that we already presented in Chapter 3. This method transforms a non-planar graph into a planar graph with dummy vertices instead of crossings. A drawing of this modified graph can be easily transformed into a drawing of the original graph by replacing the drawings of the dummy vertices by crossings.

We used the planarization method to transform the non-planar graph in our real world benchmark set into planar graphs. If the resulting graphs were not biconnected, we added some edges to make them biconnected. This step is called *planar augmentation*. A drawing of the resulting graph can be transformed into a drawing of the original graph by removing the drawings of the added edges. So we can produce a drawing of an arbitrary graph using our algorithm by applying the planarization method and planar augmentation and then transforming the resulting drawing back into a drawing of the original graph.

In more detail, we used the following approach to make graphs biconnected and planar: First, we made each of the graphs planar by deleting edges if necessary. We used the class `SubgraphPlanarizer` of the AGD-library to perform this task (see **(author?)** (AGD)). It finds a planar subgraph using PQ-trees as described by Jünger, Leipert, and Mutzel (35). Note that the resulting graph is not necessarily a maximal planar subgraph of the original graph. Then we inserted the deleted edges one by one into the planar graphs, replacing crossings we created by artificial vertices (Chapter 3 deals with this topic).

In the second step, we made the planar graphs produced by the first step biconnected. We added edges such that the resulting graph is still planar but also biconnected. Again, we used a class of the AGD to perform this task: The class `PlanAug` implements the algorithm presented by Fialko and Mutzel (21). It does not guarantee that the minimum number of edges is added to make the graph planar and biconnected but is usually very close to the optimum. For most of the graphs subjected to these planarization and augmentation steps, the number of vertices and edges increases significantly.

We applied this method to the 11529 graphs that were already used by Battista, Garg, Liotta, Tamassia, Tassinari, and Vargiu (2) to test the performance of graph drawing algorithms. The resulting set of graphs is called *real world benchmark set* in the rest of this chapter.

The second benchmark set was used by Bertolazzi, Battista, and Didimo (9) in the paper where they describe the branch & bound algorithm for computing the drawing of a biconnected planar graph with the minimum number of bends (this is the algorithm we call **B&B**). These graphs were produced using a randomized algorithm for generating biconnected planar graphs.

The algorithm starts with the triangle graph (three vertices and three edges). Two methods are used to transform the graph:

1. **Insert edge** inserts a new edge connecting two vertices if the graph remains planar after the operation and none of the two vertices has degree four. This guarantees that all the graphs that are produced are planar, biconnected, and have a maximum vertex degree of four. Since the algorithm does not check if the two vertices were already connected, the resulting graph may be a multi graph.

2. **Split edge** replaces an edge with a path of length two by inserting a new vertex.

These operations are randomly executed until the desired number of vertices and edges is reached. We call the resulting set of graphs the *artificial benchmark set.* It contains 500 graphs, 50 different graphs for each number of vertices from 10 to 100 in steps of 10.

Figure 4.21 shows how the graphs in the real world benchmark set are distributed according to their number of vertices. Most of the graphs have around 50 vertices. The largest graph had 273 vertices and 511 edges. Note that these graphs are much larger than the graphs in the original set used in (2) where the largest graphs have only 100 vertices. Additional vertices and edges were inserted during the planarization and augmentation steps to make the graphs planar and biconnected.

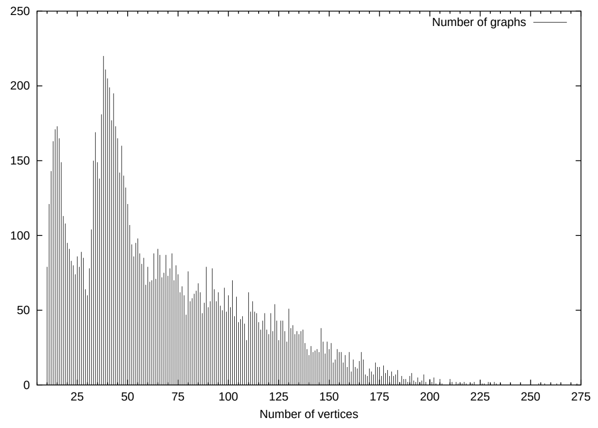


Figure 4.21: The graphs in the benchmark set broken down into sets according to the number of vertices.

One interesting statistic is the number of combinatorial embeddings for the graphs in the two benchmark sets. The computation of the number of combinatorial embeddings of a graph is simple: we just compute its SPQR-tree and multiply the number of embeddings of each skeleton. Since only R- and P-nodes have more than one embedding, we can ignore the S- and Q-nodes.

Figure 4.22 shows for each graph in the real world set its number of embeddings. There is one cross for each graph. The x-value of each cross is the number of vertices in the graph and the y-value the number of embeddings. The y-axis is logarithmic. This figure shows that the number of embeddings varies a lot for graphs of the same size because it depends strongly on the structure of the graph. If a graph of k vertices is a simple cycle, it has only one embedding. If a graph with k vertices looks like the one in Figure 4.23, it has $(k-3)!$ different combinatorial embeddings.

Figure 4.24 shows the same diagram as Figure 4.22 for the artificial benchmark set. Again, the y-axis is logarithmic. Like in the real world set, the number of embeddings for graphs of the same size varies a lot. For all numbers of vertices except 100, there is at least one graph that has only two embeddings. The graph with the most embeddings in the artificial set has 80 vertices and almost 37,748,736 embeddings, while the graph in the real world set with the greatest number of embeddings has 136 vertices and 9,437,184 embeddings. On average, the artificial graphs have more embeddings compared with the real world graphs of the same size.

We now take a look at the average number of embeddings in the benchmark sets for each number of vertices. Figure 4.25 shows this data for the real world graphs. Note that the y-axis is logarithmic again. As expected, the number of embeddings seems to grow exponentially with the size of the graphs. There are only a few graphs with high vertex number and so points scatter a lot for high x-values and the averages for graphs with more than 150 vertices are not very meaningful.

Figure 4.26 shows the same data for the artificial benchmark set. In this set, the exponential

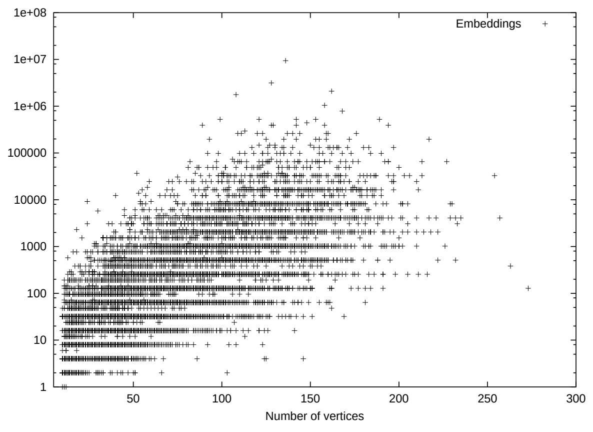


Figure 4.22: The number of combinatorial embeddings for each of the 11529 graphs in the real world set.

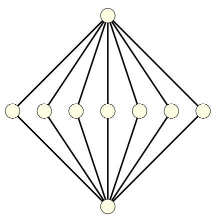


Figure 4.23: An example for a graph where the number of embeddings is exponential in the number of vertices.

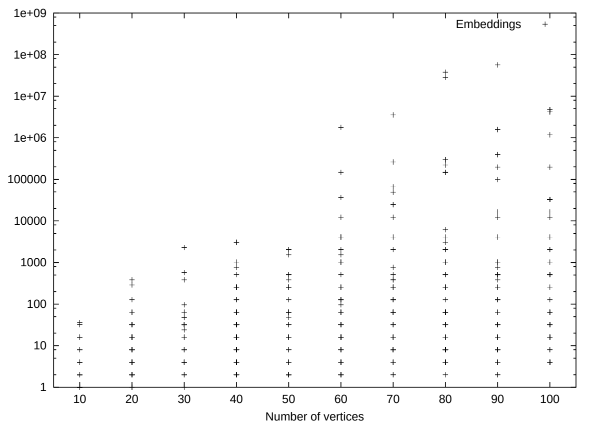


Figure 4.24: The number of combinatorial embeddings for each of the 500 graphs in the artificial set.

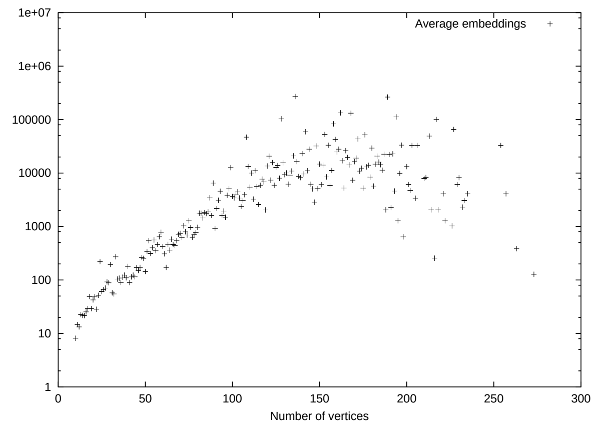


Figure 4.25: The average number of embeddings for graphs with the same number of vertices in the real world set.

growth of the number of embeddings with the size of the graphs is much more obvious.

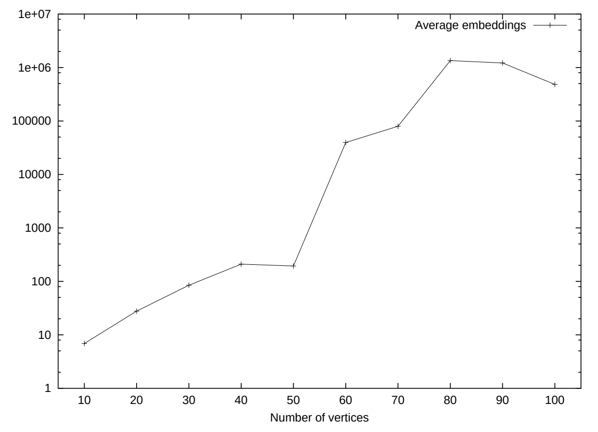


Figure 4.26: Average number of embeddings for graphs with the same number of vertices in the artificial benchmark set.

As we have already mentioned, the P- and R-nodes in the SPQR-tree of a graph determine the number of combinatorial embeddings. Therefore, we were also interested in the average number of these nodes in the graphs of the two benchmark sets.

Figure 4.27 shows the average number of P- and R-nodes of the graphs in the real world benchmark set with the same number of vertices. The number of P-nodes grows roughly linear with the number of vertices. The number of R-nodes grows until it reaches its peak at around 38 vertices and then slowly goes down until it is almost one. So large graphs have almost always only one triconnected component. The reason for this structure of the benchmark set is that the planarization method tends to turn non-planar graphs into planar graphs with big triconnected components.

Figure 4.28 shows the same data for the artificial benchmark set. In contrast to the situation for the real world graphs, both the number of P- and R-nodes grows linear with the size of the

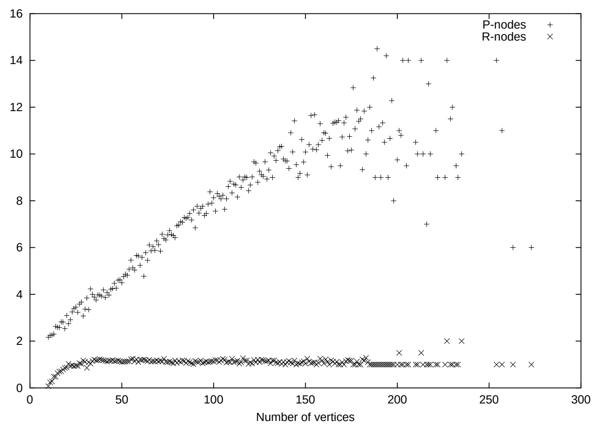


Figure 4.27: Average number of R- and P-nodes in the SPQR-trees of the graphs in the real world benchmark set.

graphs.

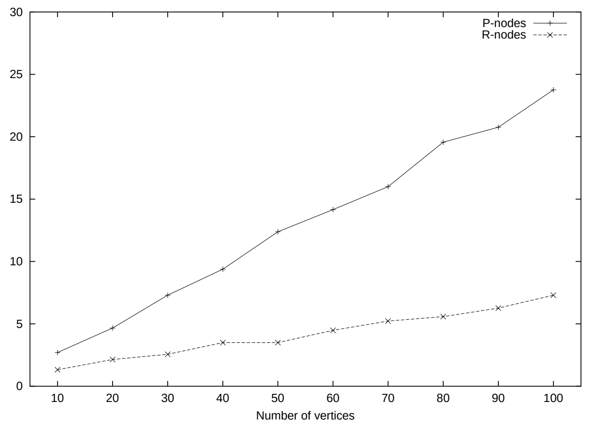


Figure 4.28: Average number of R- and P-nodes in the SPQR-trees of the graphs in the artificial benchmark set.

4.6.3 Results for the Real World Set

The first question we want to address is which of the three exact algorithms for optimizing over all embeddings is the fastest. We executed our tests on an Enterprise 450 Model 4400 with 4 GB main memory and Sun UltraSPARC-II processors running at 400 MHz.

Each of the three algorithms **B&B**, **Mix**, and **MixR** was allowed at most one hour to work on each of the graphs in the benchmark set. If the algorithm was not finished after one hour, it was terminated. Of the 11529 graphs in the real world benchmark set, **B&B** exceeded the time limit for 197 graphs, **MixR** for 87 graphs and **Mix** for 25 graphs. This already gives an indication that **Mix** has the best performance of the three algorithms.

First we compare the performance of **Mix** and **MixR**. We restricted our attention to the set of all graphs in the real world benchmark set that both algorithms were able to solve in less than one hour. For each number k of vertices, we computed the average time needed by both algorithms to compute an optimum solution. Figure 4.29 shows the diagram obtained by plotting the number of vertices on the x-axis and the average time needed by each algorithm on the y-axis. The data becomes less reliable for large graphs because there are so few of them in the benchmark set (see Figure 4.21). But it is quite obvious that **Mix** outperforms **MixR**. We will later present some additional diagrams that show why this happens.

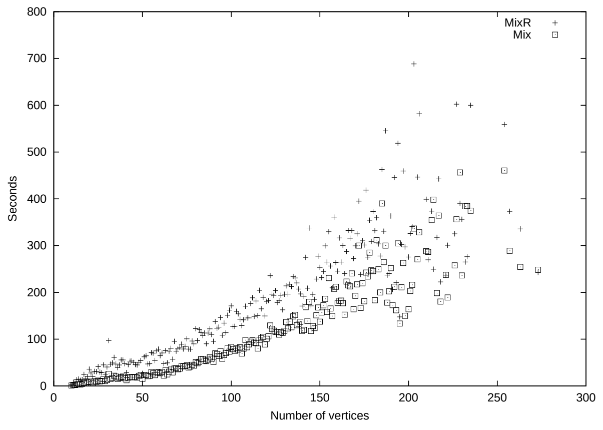


Figure 4.29: Average time needed for finding the optimum solution for graphs with the same number of vertices.

Figure 4.30 shows the average number of embeddings and the average time needed by **Mix** in the same coordinate system (note that in contrast to Figure 4.29, the y-axis is logarithmic in this figure). This diagram shows that the time needed for finding the optimum solution grows in parallel with the number of embeddings of the graph.

Figure 4.31 shows the performance comparison between **B&B** and **Mix**. Again, we computed for each number of vertices the average time needed by each algorithm. The figure shows that for graphs of up to about 120 vertices, **B&B** is faster than **Mix**. For graphs with more than 150 vertices, **Mix** seems to be faster than **B&B**. It looks like the time needed by **Mix** grows slower with the size of the graphs than the time needed by **B&B**. There are not many graphs with more than 150 vertices in the benchmark set, so the result is not as clear as we would wish. In Section 4.6.4, we will see that the advantage of **Mix** over **B&B** is much more obvious for the artificial graphs.

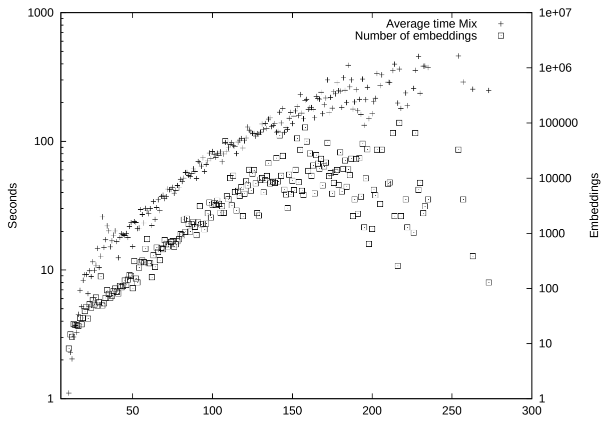


Figure 4.30: Average time needed by **Mix** and average number of embeddings drawn into the same coordinate system.

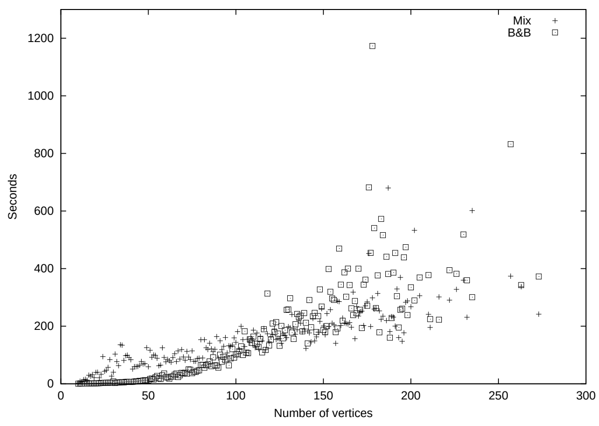


Figure 4.31: Performance comparison between **B&B** and **Mix**.

Now we want to present some statistics that explain why **Mix** performs better than **MixR**. The reason we constructed **MixR** from **Mix** is that we hoped to save constraints in the mixed integer program. The embeddings of the reduced graph are equivalent to the embeddings of the original graph but its SPQR-tree is smaller than the SPQR-tree of the original graph because some S-nodes are gone. If we only consider the integer linear program that describes all embeddings of a graph, constraints can indeed be saved by working with the reduced graph.

Figure 4.32 shows the average number of constraints in the integer linear program that describes the set of embeddings for the real world benchmark set. Note that the integer linear program that describes the embeddings of a graph is part of the mixed integer programs computed by **Mix** and **MixR**. The number of constraints grows linear with the number of vertices in the graphs. The reason for the linear growth is that the size of the SPQR-tree grows linear with the size of the graph.

Obviously, the number of constraints produced by **Mix** is always greater than the number of constraints produced by **MixR**. The reason is that the SPQR-tree used in **MixR** is smaller than the SPQR-tree used in **Mix** and the size of the embedding integer linear program depends on the size of the SPQR-tree.

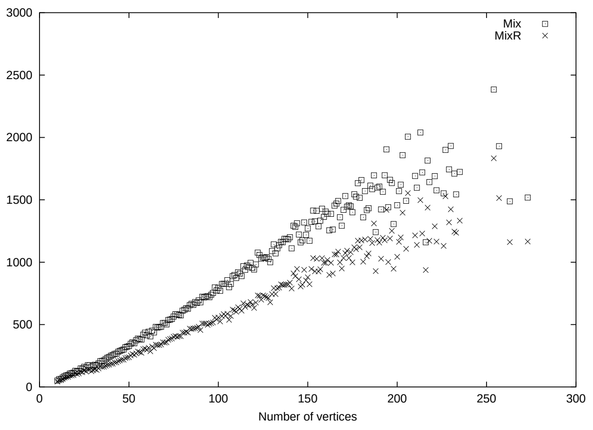


Figure 4.32: Average number of constraints in the integer linear program that describes all embeddings of the graph for the real world benchmark set.

Figure 4.33 shows the average time needed by **Mix** and **MixR** for computing the ILP describing all embeddings of the graphs. We call this time *ILP-time* in the following discussion. The ILP-time is never longer than 3.09 seconds (this can not be seen in the figure because it only shows the average for each number of vertices). For larger problems, this is not significant compared to the total time needed for computing a solution.

As we have seen in Figure 4.32, the embedding ILP of **Mix** has always more constraints than the embedding ILP of **MixR**. So we expect the ILP-time of **Mix** to be always longer than for **MixR**. This is clearly the case for the graphs with less than about 130 vertices, but then the time needed by **MixR** is often longer than the time needed by **Mix**.

Our suggestion for the reason of this strange behavior relies on two facts:

1. The function `used_time` in the LEDA-library that we use for measuring the running times is not very exact if the time measured is small.

2. The number of graphs in the benchmark set with a high number of vertices is small, so the averages we compute for high vertex numbers are not very meaningful.

To check the first of these hypotheses, we took a graph of the benchmark set with 275 vertices and 483 edges and measured for 30 runs of **Mix** the ILP-time. The ILP-time we measured using `used_time` was between 1.62 seconds and 1.82 seconds, a difference of 12.35%.

In Figure 4.21 on page 117, you can see for each number between zero and 275 the number of graphs in the benchmark set with the corresponding number of vertices. There are very few graphs with more than 200 vertices.

The greatest absolute difference between the average ILP-time for **Mix** and **MixR** was 0.9 seconds. This happens for the graph with 227 vertices (like for many numbers greater than 200, there is only one graph with 227 vertices) where **MixR** needs only 1.82 seconds and **Mix** needs 2.72 seconds to build the ILP. We expect **MixR** to build the ILP quicker than **Mix** but clearly a difference of almost 50% is rather high.

The greatest absolute time difference where **Mix** builds the ILP faster than **MixR** is 0.34 seconds (**Mix** needed 1.73 seconds to build the ILP and **MixR** 2.07 seconds). This happened for the graph with 257 vertices. Again, there is only one graph with this number of vertices, so the average is not meaningful. This is the same graph we used to test the accuracy of the `used_time` function and where we observed that the ILP-time differs by 12.35% between different runs of **Mix**. The difference between the ILP-time measured for **MixR** and **Mix** on this graph is 19%, which is not much higher than the 12.35% difference we observed on different runs of **Mix**. So the fact that the time measured for **MixR** is higher than the time measured for **Mix** does not necessarily mean that the number of computation steps executed by **Mix** in the generation of the ILP was really smaller than the number executed by **MixR**.

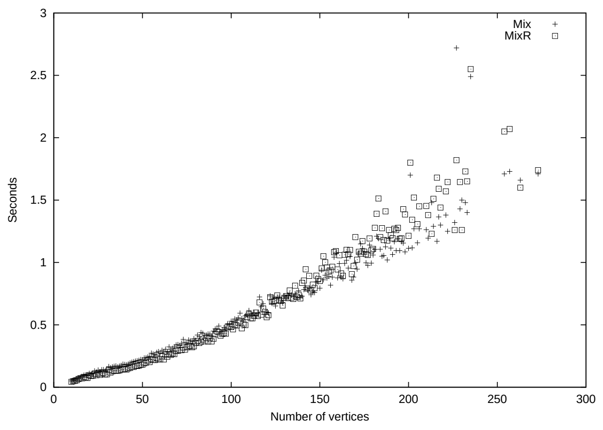


Figure 4.33: Average time in seconds needed by **Mix** and **MixR** for computing the embedding ILPs.

Figure 4.34 shows the average number of constraints in the mixed integer programs that deal with the flow network. Except for a spike at 30 vertices (the reason for this spike will be explained shortly), the number for these constraints grows linear with the size of the graph, too. But contrary to the situation for the embedding constraints in Figure 4.32, **MixR** produces more constraints than **Mix**. The reason is that the flow network for **MixR** contains more arcs than the network for **Mix**. The additional arcs are the zero-cost arcs. Since there are bounds on the amount of flow on these arcs they produce additional constraints in the mixed integer program.

Note that the number of flow constraints is much higher than the number of embedding constraints. The reason is the high number of arcs in the flow network. In any feasible solution, the flow on most of these arcs is zero because they connect cycles that are not face cycles in the embedding described by the solution.

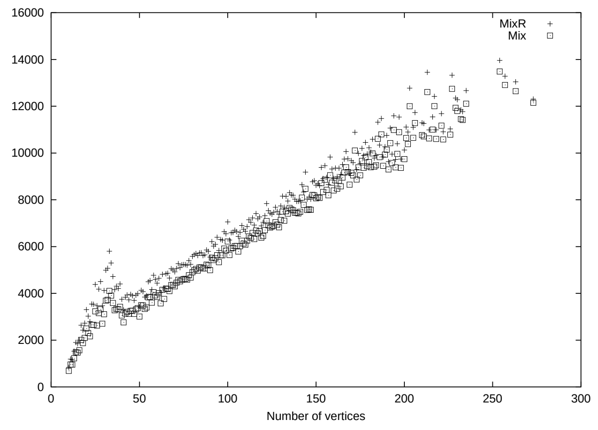


Figure 4.34: Average number of flow constraints in the mixed integer program that describes all orthogonal representations of a graph.

From ten to about 38 vertices, the number of constraints in the mixed integer linear programs grows much faster than for graphs with a higher number of vertices. The same behavior, albeit not as pronounced as for the flow constraints can also be found in the growth of the number of constraints in the ILP that describes all embeddings of the graph (Figure 4.32 on page 123) and in the number of binary variables in this ILP (Figure 4.36 on page 126).

Figure 4.27 on page 120 shows that the reason for this behavior is the structure of the graphs in the benchmark set. The figure shows the average number of P- and R-nodes of the graphs with the same number of vertices. The number P-nodes grows roughly linear with the number of vertices but also has a spike at about 38 vertices. The number of R-nodes grows until it reaches its peak also at around 38 vertices and then slowly goes down until it is almost one. So large graphs have almost always only one triconnected component.

The reason for this structure of the benchmark set is that the planarization method tends to turn non-planar graphs into planar graphs with big triconnected components. Because the number of nodes in the SPQR-tree that determine the number of embeddings grows fast up to about 38 vertices and then grows slower, the numbers of variables and constraints of our linear programs have a spike in this area. This spike is more pronounced for the flow constraints, because the flow network contains a pair of edges for each pair of cycles that could possibly be neighbors in an embedding. All these edges produce constraints in the linear program and so a small rise in the number of cycles that can be face cycles leads to a big rise in the number of flow constraints.

Figure 4.35 shows that the total number of constraints behaves exactly like the number of flow constraints because the number of embedding constraints is much smaller than the number of flow constraints. The figure shows that the number of constraints is always greater for **MixR** than for **Mix**. This is one part of the explanation why **Mix** performs better.

So we know that the mixed integer program for **MixR** has more rows than the program for **Mix**. Now we want to take a look at the number of columns or variables of the program.

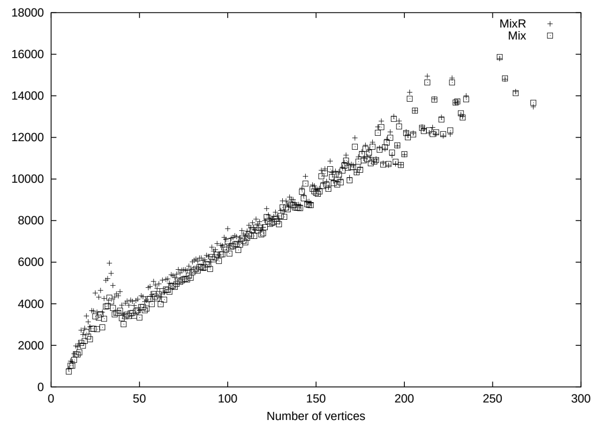


Figure 4.35: Average number of all constraints in the mixed integer programs that describe all orthogonal representations of a graph.

It is not hard to see that the reduced graph has the same number of cycles as the original graph. Only the lengths of the cycles differ. Since the integer variables in the mixed integer programs correspond to cycles in the graph, the number of integer variables is the same in the programs of **Mix** and **MixR**. Figure 4.36 shows that the number of integer variables grows roughly linear with the size of the graphs.

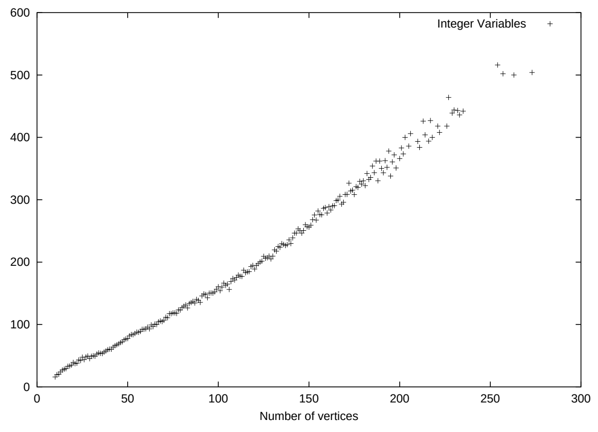


Figure 4.36: Average number of integer variables in the mixed integer programs of **Mix** and **MixR** for graphs with the same number of vertices.

Figure 4.37 shows the average number of all variables in the mixed integer programs of **Mix** and **MixR** for each number of vertices. Since the number of integer variables is the same in both programs, the difference in the number of variables is caused by the different number of continuous variables. The figure shows that the mixed integer program for **MixR** has always more continuous variables than the program for **Mix**.

Each continuous variable corresponds to the amount of flow over a specific arc in the flow network that describes the orthogonal representation. In the flow network for **MixR**, we have more arcs than in the network for **Mix**, because we introduced additional zero-cost arcs. The variables that model the flow on these arcs are responsible for the difference in the number of variables. It follows that the mixed integer program for **MixR** has more rows and more columns than the program for **Mix**.

By reducing the graph, we were able to reduce the number of constraints describing an embedding. But we have to introduce zero-cost arcs that inflate the network flow part of the mixed integer program. An increase in the number of variables and constraints does not necessarily lead to longer optimization time for mixed integer linear programs (the constraints may strengthen the formulation), but in this case it seems that the additional complexity introduced with the zero-cost arcs outweighs the reduction of complexity achieved in the embedding part of the mixed integer program.

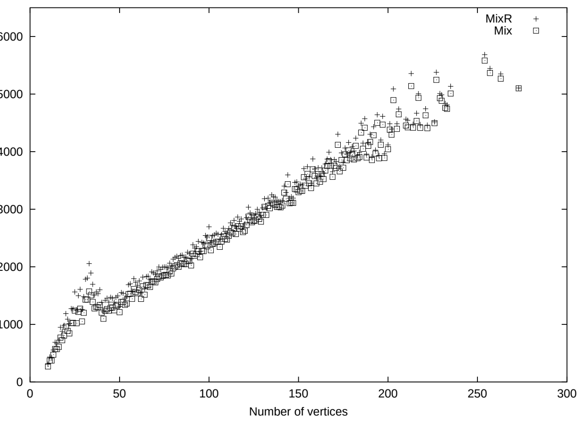


Figure 4.37: Average number of variables in the mixed integer programs that describes all orthogonal representations of a graph.

Another interesting statistic is the number of constraints that have to be separated. For the integer linear program that describes the embeddings of a P-node skeleton, we use the same formulation that is used for the asymmetric traveling salesman problem (ATSP). This formulation contains an exponential number of subtour elimination constraints. Therefore, we decided to construct a polynomial size data structure that enables the separation of these constraints.

When we have computed a solution for the MILP, we use the ATSP data structure to check for violated subtour elimination constraints. If we find a violated subtour elimination constraint, we add it to the MILP and re-optimize. Note that this can only happen if there is a P-node skeleton in the SPQR-tree of the graph that has more than four edges. Otherwise, we add all necessary inequalities to the program from the start.

When we applied the **Mix** algorithm to all the graphs in the benchmark set, the separation of constraints was only necessary for six graphs (remember that there are 11529 graphs in the set). We never had to separate more than one constraint. As already mentioned on page 104, we have hard-coded a complete polyhedral description for the case that a P-node skeleton has four edges into our algorithm. Since there are few P-nodes in the SPQR-trees for graphs in the benchmark set where the skeleton has more than four edges, this almost eliminates the need for separation of subtour elimination constraints.

Figure 4.38 shows the average running time of **Heuristic** for the graphs in the real world benchmark set. The running time grows faster than linear because a shortest path has to be computed in the extended dual graph of the input graph. However, **Heuristic** is very fast and takes just 1.26 seconds for the largest graph in the set.

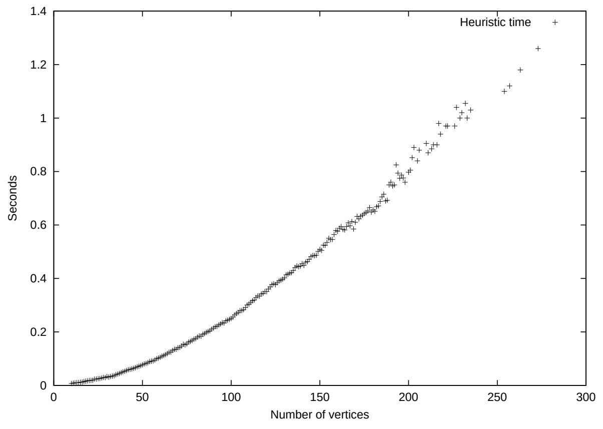


Figure 4.38: Average time needed by **Heuristic** to compute an orthogonal representation for the graphs in the real world benchmark set.

The next question is: How many bends can we save if we optimize over all embeddings instead of choosing an arbitrary embedding? So we applied the heuristic to all graphs in our benchmark set and compared the number of bends with the optimum solutions.

Let h be the number of bends in the orthogonal representation computed by the heuristic and o be the number of bends in an orthogonal representation with the minimum number of bends. For each graph in the benchmark set, we computed the following value:

$$\frac{h-o}{h}100\%$$

This is the percentage of the improvement we get using an optimal algorithm. We broke the set of all graphs into ten subsets. The first subset containes the graphs where the improvement is smaller than ten percent, the second subset the graphs with improvement between ten and 20% and so on. The last subset containes the graphs where the improvement is between 90 and 100%. Note that the improvement is 100% if the heuristic solution has bends while the optimum solution contains no bends. Figure 4.39 shows the corresponding diagram. The x-axis shows the improvement ranges while the height of the boxes corresponds to the number of graphs that fall into that range.

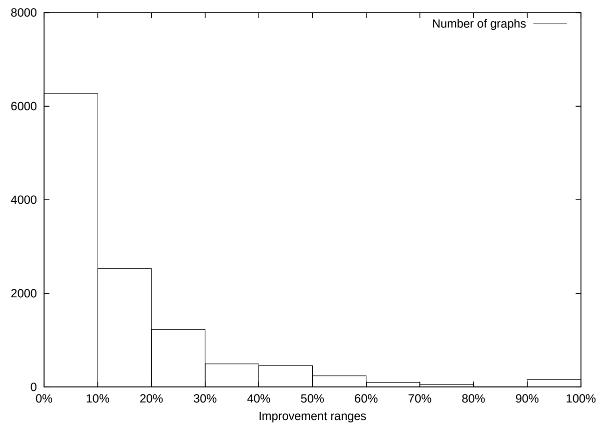


Figure 4.39: Number of graphs broken down into improvement ranges.

Exactly 5224 graphs had an improvement greater than ten percent. So almost half of all graphs show significant improvements. Only 1354 graphs (about 12% of all graphs) show no improvement at all. The greatest absolute difference in the number of bends that we observed from the heuristic solution to the optimal solution was 16 bends. The graph in question had 130 vertices and 205 edges. The heuristic computed an orthogonal representation with 48 bends while the optimum solution had 32 bends (see Section 4.6.5 on page 133 for the corresponding drawings). The average number of saved bends per graph using the optimal algorithm was 3.8. The average improvement over all graphs was 21.3%.

So the improvements achieved using the optimal algorithm compared to the heuristic are significant. The main drawback is that solving the problem to optimality can be very time consuming. The longest computation time for one graph that we observed for the heuristic was one 1.26 seconds. On the other hand, there were 25 graphs that our fastest exact algorithms was not able to solve in one hour. The average time needed per graph by the **Mix** algorithm was just 56 seconds (excluding the 25 graphs that take more than one hour).

In many applications, the quality of a drawing is very important and the time needed to generate the drawing is of secondary importance. For example, if the graph drawing will be used as an illustration in a book, it should be no problem to wait a few minutes for the finished drawing. For these applications, our optimal method is better suited than the heuristic (if the graph is not too big) since it decreases the number of bends on average by about 20% compared to the heuristic.

4.6.4 Results for the Artificial Benchmark Set

The results for the real world benchmark set have shown that **Mix** performs better than **MixR**. Therefore, we only give results for **Mix**, **B&B**, and **Heuristic** for the artificial benchmark set.

We start with the comparison of the time needed by **Mix** and **B&B** for finding an optimal solution. Figure 4.40 shows the average time needed by the two algorithms for each number of vertices. The performance advantage of **Mix** is very obvious. **B&B** needs almost twice as long as **Mix** on average to find an optimal solution. While **Mix** failed to provide the solution in one hour for only one of the 500 graphs, **B&B** failed for 43 graphs.

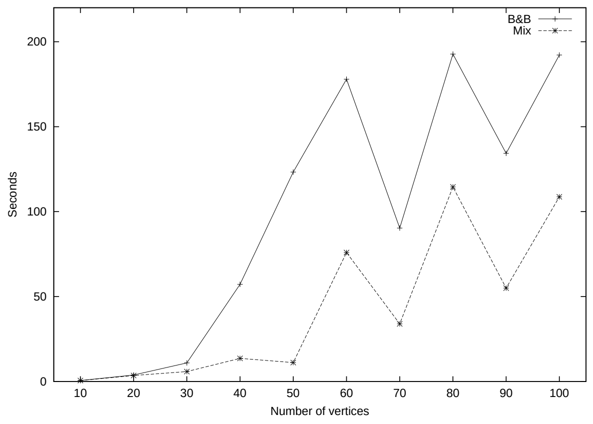


Figure 4.40: Average time needed by **B&B** and **Mix** for finding the optimum solution for graphs with the same number of vertices.

Figure 4.41 shows the average time needed by the heuristic to compute orthogonal representations for the graphs in the artificial benchmark set. This time grows faster than linear (because a minimum cost flow must be computed), but for the graphs with 100 vertices, the average time is still less then a quarter of a second. These running times are negligible compared to the running times of **Mix** and **B&B**.

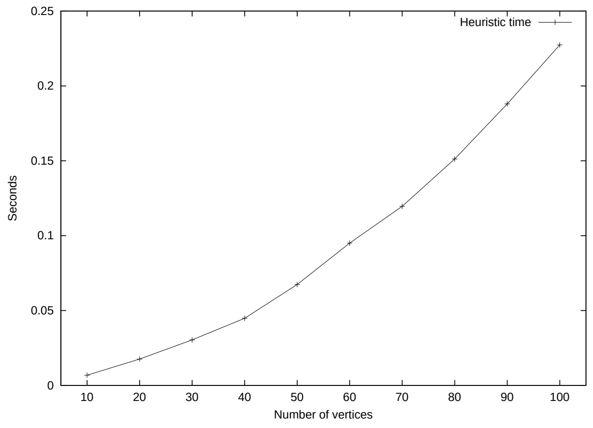


Figure 4.41: Average time needed for **Heuristic** to compute an orthogonal representation for the graphs in the artificial benchmark set.

Figure 4.42 shows for each number of vertices the average number of constraints in the ILP computed by **Mix** that describes all embeddings of a graph and the time needed to compute these constraints. The left hand scale shows the time in seconds while the right hand scale shows the number of constraints. It seems that the number of constraints grows roughly linear with the size of the graph. Except for a bump at 90 vertices, the time needed for computing the ILP grows parallel with the number of embedding constraints.

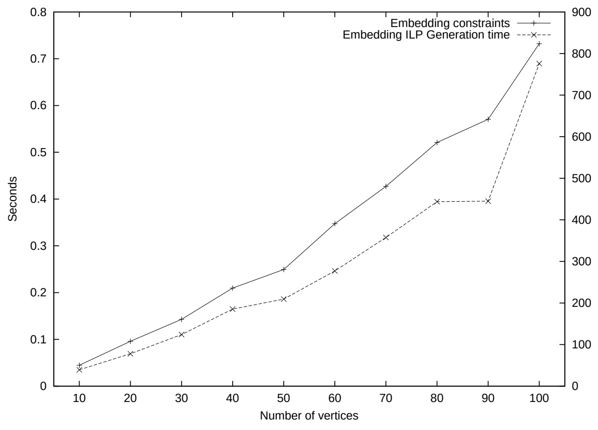


Figure 4.42: Average number of constraints in the integer linear program (ILP) that describes all embeddings of the graph and the average time needed to compute the ILP.

Figure 4.43 shows for each number of vertices the average number of integer variables in the mixed integer linear program (MILP) computed by **Mix**. The average grows linear with the size of the graph and for the graphs with 100 vertices, we needed on average less than 140 binary variables. This does not seem very high if we compare it with the average number of embeddings of the graphs with 100 vertices, which is almost half a million.

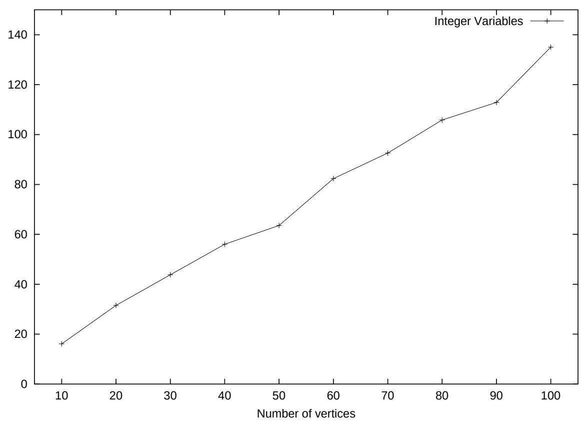


Figure 4.43: Average number of integer variables in the mixed integer programs of **Mix** for graphs with the same number of vertices.

The average number of all variables and constraints in the MILP computed by **Mix** is shown in Figure 4.44. Both averages grow in lockstep and roughly linear with the size of the graphs.

Mix did not separate any subtour elimination constraints, because the maximum vertex degree of the graphs in the artificial benchmark set is four. It follows that their SPQR-trees do not contain P-node skeletons with more than four edges. The separation of subtour elimination constraints can only be necessary for graphs that contain P-node skeletons with at least five edges because we use complete descriptions of the embedding ILPs for the smaller P-node skeletons.

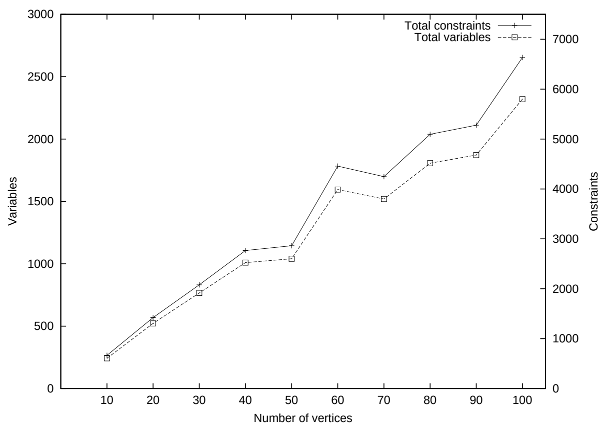


Figure 4.44: Average number of total variables and constraints in the mixed integer linear programs computed by **Mix** for the artificial benchmark set.

Finally, we examine the improvements in the quality of the solutions achieved by using exact algorithms instead of the heuristic for the artificial graphs. We define the improvement in the same way as for the real world graphs. If h is the number of bends in the orthogonal representation computed by **Heuristic** and o the number of bends in an orthogonal representation computed by an exact algorithm (**Mix** or **B&B**), then we define the improvement as

$$\frac{h-o}{h}100\%$$

which is the percentage of the improvement we get using an exact algorithm. We split the set of 500 graphs into ten sets according to the range of their improvement. Figure 4.45, that shows the corresponding diagram, is very similar to Figure 4.39 on page 129, where the improvement ranges for the real world graphs are shown. For almost half of the graphs we achieved an improvement greater than 10 percent.

The maximum absolute difference we observed was 12 bends for a graph with 100 vertices and 160 edges. The solution found by the heuristic had 54 bends while an optimal solution has only 42 bends. There are 18 graphs where the optimum solution has no bends at all and the heuristic solution contains bends.

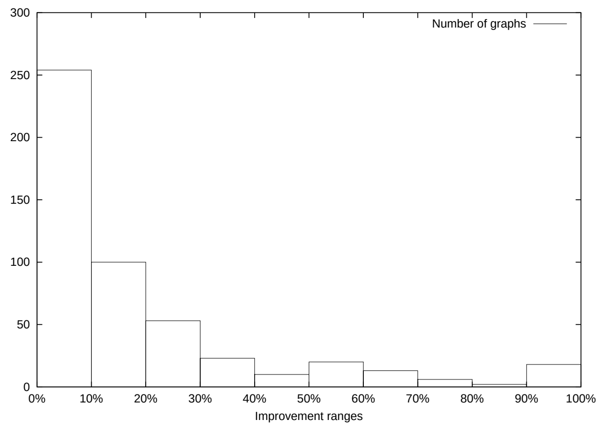


Figure 4.45: Number of graphs in the artificial benchmark set broken down into improvement ranges.

4.6.5 Two Example Graphs

After presenting the big statistic picture, we now present the results for two example graphs of the benchmark set and show the drawings produced by GDToolkit and our new algorithm.

GDToolkit contains a demo program called `GRID`. This is a graph editor that can also call most of the algorithms in the GDToolkit library and display the drawings produced by them. The implementation of our new algorithm uses `GraphWin`, the graph editor of the `LEDA` library (see `http://www.mpi-sb.mpg.de/LEDA`) as a graphical user interface and for displaying the drawings.

We present the drawings computed by three different algorithms. From the GDToolkit library, we used **Heuristic** and **B&B**. The third algorithm is **Mix** that we implemented using the `LEDA` library.

We have chosen two graphs out of the real world benchmark set where the solution computed by the two optimal algorithms is significantly better than the solution computed by **Heuristic**. As we have already mentioned, we made the graphs from the real world benchmark set planar using the planarization method and then added new edges in an augmentation step to make the graphs biconnected. The first graph we use in this section is derived from the graph called `grafo596.14`. The original version of this graph is already planar, and we only had to add two edges to make it biconnected and planar. The resulting graph, which we call `small`, has 14 vertices and 18 edges.

The second graph is derived from the graph `grafo10131.98`. This graph is not planar and not biconnected, has 98 vertices and 126 edges. We first used the planarization method on the graph and then added 13 edges to make it biconnected and planar. The resulting graph has 130 vertices and 205 edges and we call it `BIG`.

The following table shows the number of bends in the orthogonal representations computed by the three algorithms for `small` and `BIG`.

	Heuristic	**B&B**	**Mix**
`small`	4	0	0
`BIG`	48	32	32

It is not surprising that the two optimal algorithms produce the same number of bends. As we have already mentioned, we have chosen examples where **Heuristic** performs badly. For the graph `small`, it produced four bends while an optimum solution has no bends at all. For the graph `BIG`, it produces 16 bends more than necessary. The next table shows the running times in seconds for the three algorithms and the two graphs.

	Heuristic	**B&B**	**Mix**
`small`	0.01	0.42	0.3
`BIG`	0.42	5243.91	108.14

Heuristic is extremely fast compared with the optimal algorithms. Even for the `BIG` graph, it needs only a fraction of a second. The graph `small` is somewhat atypical, because usually **B&B** is faster on small graphs than **Mix**. But the difference is not significant. The difference in the running time for the graph `BIG` is very significant. **B&B** needs almost 1.5 hours while **Mix** needs less than two minutes. As the computational results in the last section have shown, **Mix** is in general faster than **B&B** for large graphs but the difference is not usually as extreme as for the graph `BIG`.

For both graphs, **Mix** does not have to separate any subtour elimination constraints. We have inspected the two mixed integer linear programs (MILPs) computed by **Mix** and the data CPLEX provided about the computation that produced the optimum solution.

The MILP for the graph `small` has 587 rows and 222 columns. The number of binary variables is 24. These variables are the 16 cycle variables and the eight outer face variables. CPLEX was able to eliminate 233 rows in its preprocessing step, but it could not eliminate any columns. The resulting matrix has 1,110 non-zeroes. For this small problem, the branch & cut tree has only three nodes. The total number of simplex iterations was 163.

One flow cover cut was used in the root node. This type of inequalities can be used for mixed integer linear programs that describe a flow in a network where the capacity of arcs depend on the value of binary variables. This is the case for our program because the arcs that are incident to a node in the network representing a cycle in the original graph have zero capacity if the cycle is not chosen as a face cycle. Flow cover cuts were introduced by Padberg, Roy, and Wolsey (45). Gu, Nemhauser, and Savelsbergh (26) introduced an efficient way of lifting the inequalities.

The MILP for the graph `BIG` has 10,059 rows and 3,473 columns. The number of binary variables is 333. Of these variables, 222 are cycle variables and 111 are outer face variables. For this problem, CPLEX was able to eliminate 4,920 rows and 544 columns in the preprocessing step. The resulting matrix has 15,974 non-zeroes. The branch & cut tree has 282 nodes and 32,512 simplex iterations were needed to compute the solution. The number of flow cover cuts generated in the root node was 20. Note that these cuts are valid for every node in the tree.

Figure 4.46 shows the drawing of `small` computed using **Heuristic** while Figures 4.47 and 4.48 show the same graph drawn using **B&B** and **Mix**, respectively. Note that the reproduction of the drawings produced using GDToolkit (**Heuristic** and **B&B**) is not as good as the reproduction of the drawings that were produced using our new algorithm, because the `GraphWin` graph editor we used to call our algorithm can export a drawing as a postscript file, while the GRID editor of GDToolkit does not have this feature. Therefore, we had to use screen grabs to produce a postscript file, which has a negative influence on the quality of the resulting pictures.

Note that **B&B** and **Mix** have produced the same embedding for the graph `small`, but they have found different orthogonal representations. The drawing produced by **Heuristic** realizes a different embedding. This follows already from the fact that the drawings computed by **Heuristic** have the minimum bend number for the chosen embedding.

Figure 4.49 shows the drawing of the graph `BIG` produced using **Heuristic**, while the Figures 4.50 and 4.51 show the drawings produced using **B&B** and **Mix**. Observe that the two optimal algorithms have chosen different embeddings for the graph `BIG`. This can be seen by looking at the sequence of the edges around each vertex in clockwise order. If we look for example at vertex number 40 in Figure 4.50, than the sequence of the edges around it is 46,93,18,126 (we identify the edges by the number of the other incident vertex). In Figure 4.51, this sequence is 46,18,126,93. So there are at least two different embeddings of `BIG` for which we can produce an orthogonal representation with the minimum number of bends.

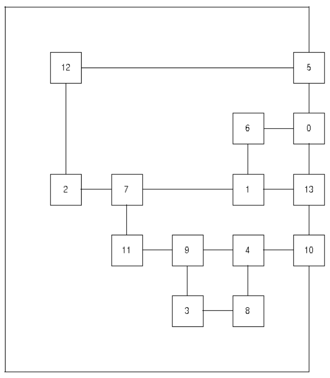


Figure 4.46: Drawing of the graph `small` produced using **Heuristic**.

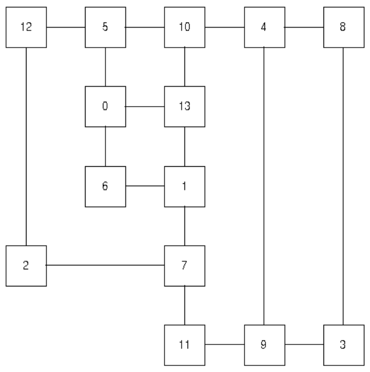


Figure 4.47: Drawing of the graph `small` produced using **B&B**.

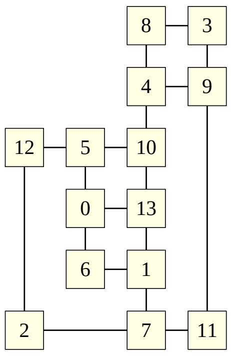


Figure 4.48: Drawing of the graph `small` produced using **Mix**.

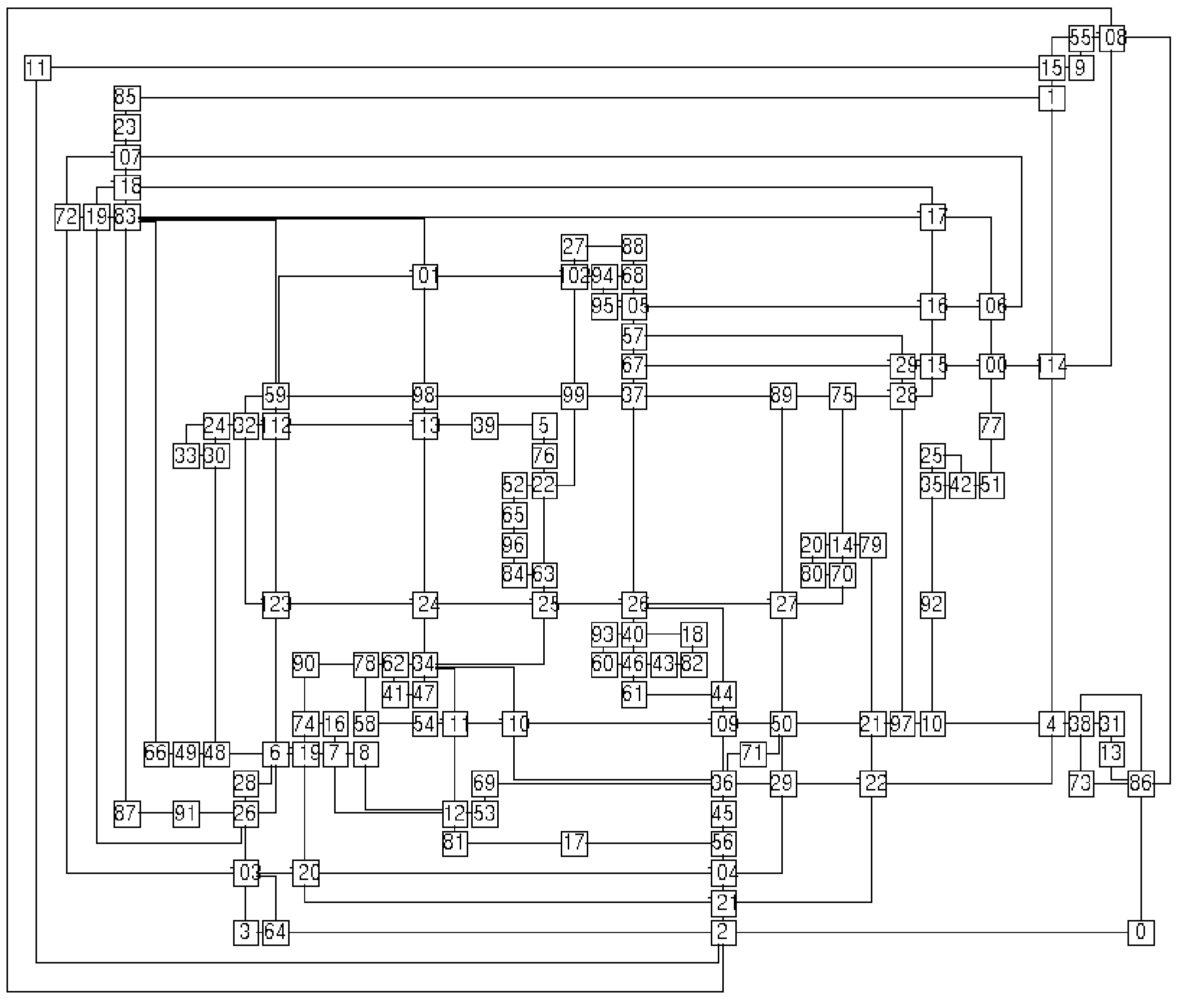


Figure 4.49: Drawing of the graph `BIG` produced using **Heuristic**.

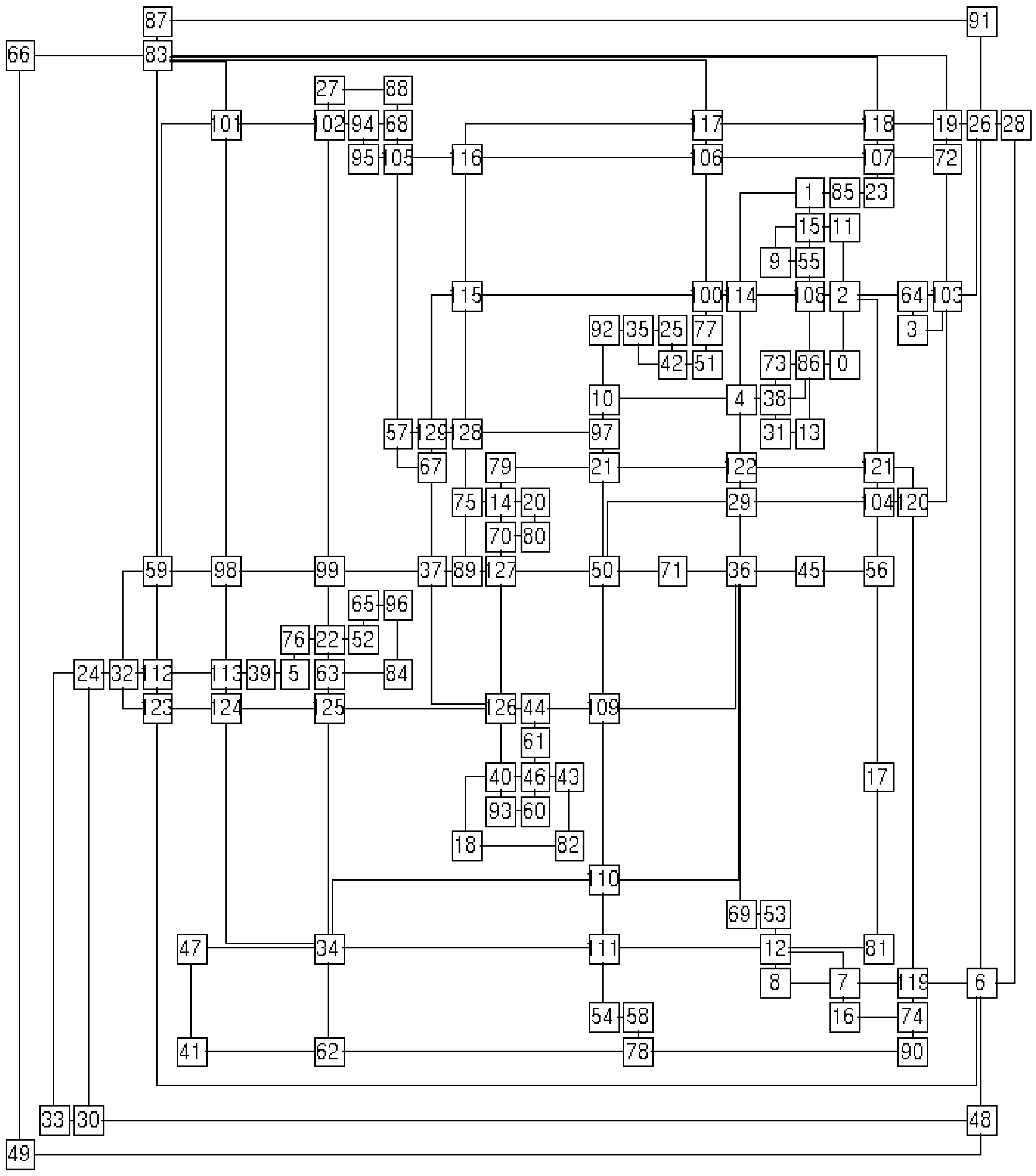


Figure 4.50: Drawing of the graph BIG produced using **B&B**.

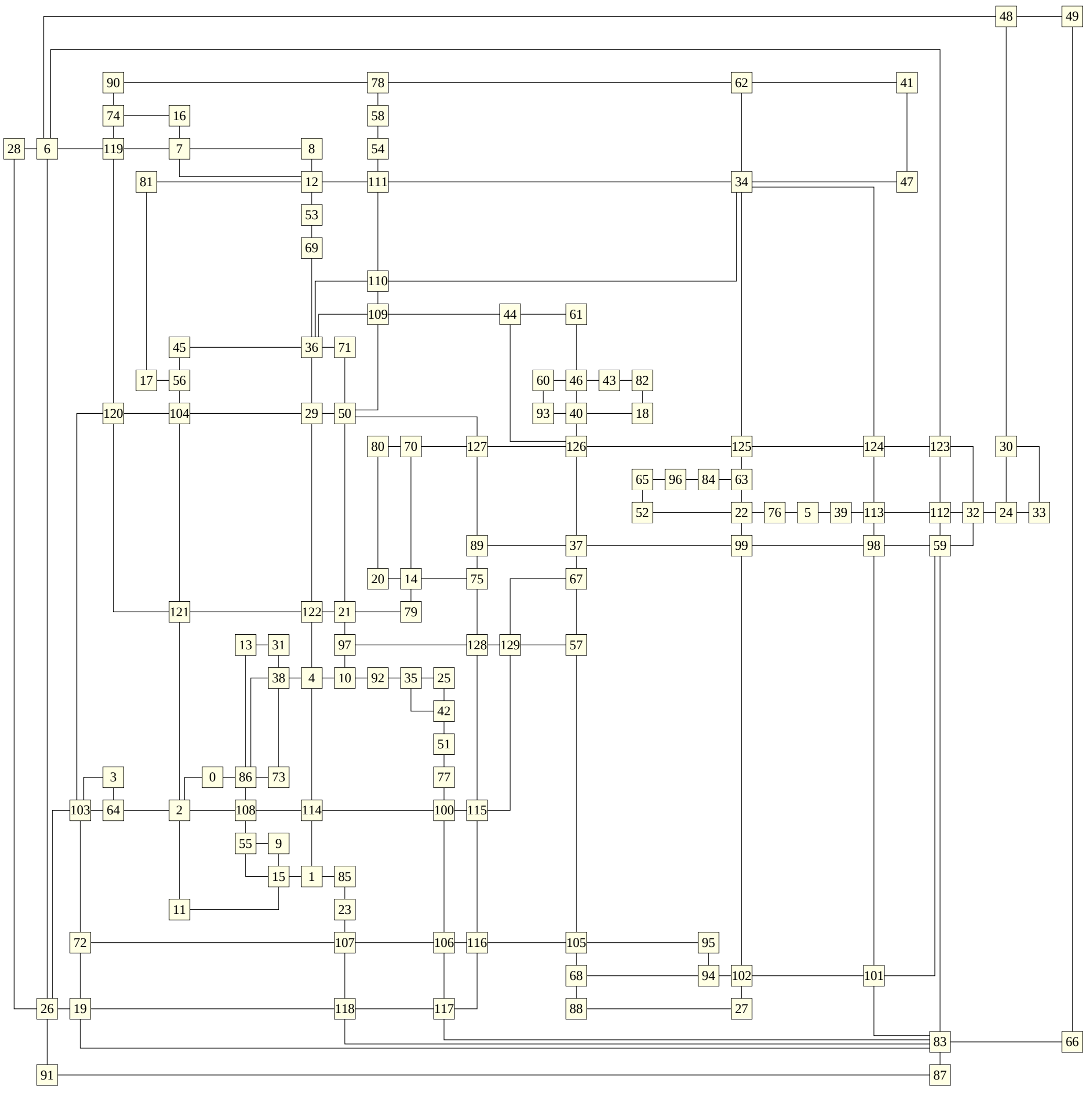


Figure 4.51: Drawing of the graph BIG produced using **Mix**.

Chapter 5

Conclusion

In this thesis, we present two new applications for SPQR-trees in the field of graph drawing. Both use the fact that the SPQR-tree of a planar biconnected graph can enumerate all embeddings of the graph. We use two different approaches to take advantage of the properties of the SPQR-tree for solving problems where the right choice of the embedding for a graph is important.

Both problems fit into the following framework: We have a function $f : P \rightarrow \mathbb{N}_0$ where P is the set of embeddings of a certain graph. This function assigns a non-negative integer value to each embedding. We are looking for an embedding Π where the value of f is minimal.

For both problems that we are looking at, the value of the function f can be computed in polynomial time for a fixed Π. Since triconnected graphs have only two different embeddings that are mirror images of each other, we can also compute the value of f for the embeddings of a triconnected graphs in polynomial time.

Our first approach works as follows: We use the SPQR-tree to compute the triconnected components of the input graph G. We modify each of the components slightly such that computing the value of f for each component and adding the resulting values results in the minimum value f_{min} for f over all embeddings of G. In the process, we also compute additional information that enables us to find in polynomial time an embedding Π such that $f(\Pi)$ is f_{min}.

The problem that we attacked with this approach is finding an embedding for a planar graph such that a certain edge can be inserted into the embedding with the minimum number of crossings. So the function f assigns to each embedding the number of crossings produced when the additional edge is inserted into the embedding.

The second approach we use is more complex. We use it for a function f where the decision problem if there exists an embedding Π of G where $f(\Pi)$ is smaller than k for a certain k is NP-complete. Again, we want to compute the minimum value f_{min} over all embeddings of G. For the problem we are dealing with, there is a linear program LP where the optimal solution is $f(\Pi)$ for a fixed embedding Π. We develop an integer linear program ILP where the set of feasible solution corresponds exactly to the set P of embeddings of G. This program is computed recursively using the structure of the SPQR-tree of G. By combining ILP and LP, we get a mixed integer linear program where the optimal solution has value f_{min}.

We used this approach for the problem of finding an orthogonal drawing with the minimum number of bends for a planar biconnected graph G. There is a linear program for computing the minimum number of bends necessary to draw G for a fixed embedding. We combined it with our new integer linear program where the solutions correspond to the embeddings of the graph. An optimal solution of the resulting mixed integer linear program corresponds to an orthogonal representation of the graph with the minimum possible number of bends over all embeddings.

5.1 Inserting an Edge with the Minimum Number of Crossings

Each instance of this problem consists of a planar graph $G = (V, E)$ and a pair of vertices $e = (v_1, v_2)$ with $e \notin E$. The goal is to find a drawing of $G' = (V, E \cup \{e\})$ with the minimum number of crossings and the property that edges in E do not cross each other. This problem is part of the planarization method for drawing non-planar graphs.

If the embedding is fixed, the solution for the problem is quite simple: We construct the geometric dual of G with respect to the given embedding, add the vertices v_1 and v_2 and connect them to all the vertices in the dual graph that represent incident faces. Then we only have to find a shortest path from v_1 to v_2 in the resulting graph. The edges used by the shortest path are the duals of the edges that have to be crossed in an optimal solution to the edge insertion problem.

Our new algorithm computes the optimal solution even when the embedding is not part of the input. We first compute the SPQR-tree for G. Then we compute the set of R-nodes that determine the number of edges to be crossed. The R-nodes of the SPQR-tree correspond to subdivisions of triconnected graphs contained in G. In the skeletons of these nodes, we insert new vertices that represent v_1 and v_2 if necessary. Then we replace some edges in the skeleton with the subgraphs of G that they represent. On the resulting graph, we use the algorithm for solving the problem in the case of a fixed embedding.

The concatenation of all the edge lists computed for each of the R-nodes by the algorithm is the list of edges that have to be crossed by e in an optimal solution for the edge insertion problem. The size of the SPQR-tree data structure is linear in the size of G and can be computed in linear time. Our algorithm refers to each edge in the SPQR-tree and each edge in G only a constant number of times. It follows that the running time for the algorithm is linear in the size of G.

Our computational results show that using our new algorithm in the planarization method reduces the number of crossings in the resulting drawings significantly compared to using the heuristic that has formerly been used. On a benchmark set of real world graphs, the percentage of crossings saved using the new method is 14% on average.

5.2 Minimizing the Number of Bends in an Orthogonal Drawing

In an orthogonal drawing of a planar graph, all the edges are drawn as sequences of horizontal and vertical line segments. The points where two segments of the same edge meet are called bends. The decision problem if there is an orthogonal drawing of a certain graph with less than k bends is known to be NP-complete.

Our new exact algorithm for solving the problem has two main components. The first component is a linear program that is constructed from a graph G and an embedding Π. An optimum solution for the program corresponds to an orthogonal representation of G that realizes Π and has the minimum number of bends. An orthogonal representation is a data structure that describes the topology of an orthogonal drawing and the sequence of the bends on each edge but does not fix the lengths of the edge segments that constitute the edges. The linear program is derived from a previously known formulation of the problem as a minimum cost flow network.

The second ingredient is our new integer linear program that can be constructed for any planar biconnected graph. The set of solutions of this program corresponds exactly to the set of embeddings of the graph. The program is constructed recursively using the SPQR-tree of the graph.

We combined our new integer linear program that describes the embeddings of a graph with the linear program that describes the orthogonal representations of a graph for a fixed embedding to construct a mixed integer linear program where an optimal solution corresponds to an orthogonal representation of the graph with the minimum number of bends among all embeddings. The result is a mixed integer linear program where an optimum solution corresponds to an orthogonal representation of the problem graph with the minimum number of bends among all possible orthogonal representations realizing any embedding of the graph.

We solve this mixed integer linear program using the commercial mixed integer solver CPLEX. We compared the performance of this approach with a known branch and bound approach designed to solve the same problem. For real world graphs of up to about 120 vertices, the branch and bound approach is faster. For larger graphs, our new approach outperforms the branch and bound approach. For artificially generated graphs, that were already used in the literature for testing the performance of the branch & bound approach, our new method is almost twice as fast as the Branch & Bound method.

5.3 Possible Starting Points for Future Research

First we take a look at the area of crossing minimization. In this thesis, we presented a linear time algorithm that inserts one edge into a planar graph with the minimum number of crossings under the condition that the edges of the original planar graph do not cross in the solution.

It is an interesting problem if this can be done in polynomial time for more than one edge. A description of the problem is as follows: Given a planar graph $G = (V, E)$ and a set L of pairs of vertices of G where $L \cap E = \emptyset$, find a drawing of $G' = (V, E \cup L)$ with the minimum number of crossings under the condition that edges in E do not cross.

This problem can not be solved by iteratively calling our algorithm for all edges in L because the number of crossings generated in this way depends on the sequence of the insertions of the edges in L. To see this, we can use the same example already used in Section 3.6.

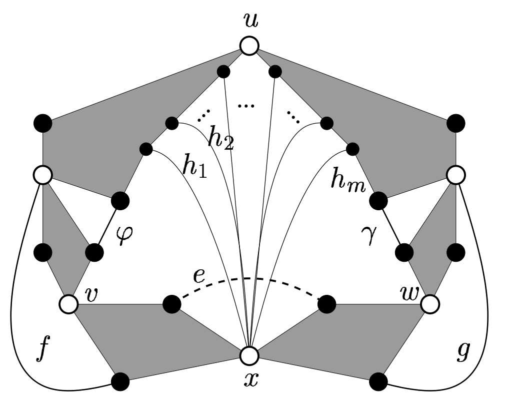


Figure 5.1: If we first insert first the edges f and g, the insertion of e generates m crossings.

We can assume that Figure 5.1 was constructed inserting the three edges e, f, and g iteratively. We first inserted f and g with the minimum number of crossings. In both cases, no crossings were generated. But then the insertion of edge e generates m crossings.

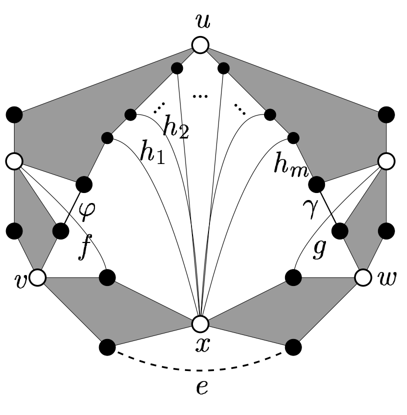


Figure 5.2: If we insert the edge e first and then the edges f and g, we only generate two crossings.

We can interpret Figure 5.2 as the result of first inserting edge e and then the edges f and g. This sequence of insertions produces only two crossings. It follows that any algorithm that inserts more than one edge with the minimum number of crossings into a graph has to consider all edges that have to be inserted at once.

In the field of bend minimization, an extension of our algorithm to connected graphs would be a worthwhile endeavor. Until now, there is no algorithm known that finds a drawing with the minimum number of bends for a planar graph over all embeddings if the graph is not biconnected.

One heuristic is to add dummy edges to the graph to make it biconnected and then assigning zero costs to bends on the dummy edges for the optimization. But this still does not guarantee that the resulting orthogonal representation has the minimum possible number of bends after deleting the dummy edges. The reason is that inserting dummy edges decreases the number of possible embeddings. So it may happen that the embeddings of the original graph that give the minimum number of bends can not be realized when the dummy edges are added.

An optimal algorithm for the problem will probably work by splitting the graph into its biconnected components. Then we have to find orthogonal representations with the minimum number of bends for the components under the condition that we can later merge the resulting orthogonal representations into an orthogonal representation of the original graph. This means that some information about the structure of the whole graph have to be used in the optimization of the components.

Appendix A

Deutsche Zusammenfassung

Ein Graph ist eine mathematische Struktur, die oft benutzt wird, um die Beziehungen zwischen verschiedenen Objekten darzustellen. Ein Graph besteht aus einer Menge von *Knoten* und einer Menge von Paaren dieser Knoten, den *Kanten*.

Man kann einen Graphen in Form einer zwei-dimensionalen booleschen Matrix angeben, bei der der Eintrag (v_1, v_2) auf “wahr” gesetzt ist, wenn die Kante (v_1, v_2) in der Kantenmenge enthalten ist. Auch kann man einen Graphen als Liste seiner Kanten angeben. Aber in den meisten Fällen ist die in einem Graphen gespeicherte Information für einen Menschen am einfachsten erschließbar, wenn der Graph durch eine Zeichnung dargestellt wird.

Meistens wird ein Graph wie folgt gezeichnet: Die Knoten werden als Formen wie z.B. Kreise oder Rechtecke gezeichnet während die Kanten als Linien dargestellt werden, die diese Formen verbinden. Ein Punkt, an dem sich zwei dieser Linien schneiden wird als *Kreuzung* bezeichnet, wenn es sich nicht um die Darstellung eines Knotens handelt, der in den beiden betroffenen Kanten vorkommt. Ein Graph heißt *planar*, wenn man ihn ohne Kreuzungen in die Ebene zeichnen kann. Eine Zeichnung ohne Kreuzungen nennt man entsprechend eine *planare* Zeichnung.

Die Menge der verschiedenen planaren Zeichnungen eines planaren Graphen ist unendlich groß, aber man kann diese Menge in endlich viele Äquivalenzklassen einteilen, die man *Einbettungen* des Graphen nennt. Ein Graph *realisiert* eine bestimmte Einbettung Π, wenn er zu der Äquivalenzklasse Π gehört. Alle planaren Zeichnungen, die die gleiche Einbettung realisieren, sind in einem topologischen Sinne äquivalent.

Die Datenstruktur *SPQR-Baum* repräsentiert die Zerlegung eines zweizusammenhängenden Graphen in seine dreizusammenhängenden Komponenten. Die Menge der Einbettungen einer jeden Komponente ist leicht aufzählbar und die Kombination der Einbettungen aller Komponenten ergibt eine Einbettung des Originalgraphen. Deshalb kann man einen SPQR-Baum dazu benutzen, alle Einbettungen eines planaren zweizusammenhängenden Graphen aufzuzählen. Legen wir nur die Einbettungen eines Teiles der Komponenten des SPQR-Baums fest, so können wir eine Menge von Einbettungen des Originalgraphen darstellen.

Der Umstand, der den SPQR-Baum so interessant macht, ist dass es viele interessante Probleme auf dem Gebiet der planaren Graphen gibt, die für Graphen mit einer festgelegten

Einbettung einfach zu lösen sind, aber schwieriger werden, wenn die Einbettung nicht gegeben ist. Diese Probleme sind ebenfalls einfach zu lösen, wenn der Graph dreizusammenhängend ist. Der Grund hierfür ist, dass dreizusammenhängende Graphen nur zwei verschiedene Einbettungen besitzen, die Spiegelbilder voneinander sind. Im Gegensatz dazu steigt bei zweizusammenhängenden Graphen die Größe der Menge von Einbettungen exponentiell mit der Größe des Graphen.

Nehmen wir nun an, wir wollen ein Problem für zweizusammenhängende Graphen lösen, das einfach ist, wenn eine Einbettung gegeben ist. Es folgt daraus, dass wir das Problem auch einfach für die dreizusammenhängenden Komponenten im SPQR-Baum des Graphen lösen können. Falls es uns gelingt, einen Weg zu finden, die Lösungen für die Komponenten des Graphen zu einer Lösung für den gesamten Graphen zu kombinieren, so haben wir einen Algorithmus gefunden, der unser Problem für zweizusammenhängende Graphen löst, wenn die Einbettung nicht Teil der Eingabe ist.

Wir haben diesen Ansatz benutzt, um das folgende Problem zu lösen: Gegeben ist ein planarer Graph G und eine zusätzliche Kante e, die nicht in G enthalten ist. Wir bezeichnen den Graphen, der durch Einfügen von e in G entsteht mit G'. Wir suchen eine Zeichnung für G' mit möglichst wenigen Kreuzungen unter der Bedingung, dass Kanten von G sich nicht kreuzen. Dieses Problem tritt in der *Planarisierungs Methode* zum Zeichnen von nicht-planaren Graphen auf.

Falls eine Einbettung von G gegeben ist, in die wir die Kante e einfügen sollen, so können wir das Problem schnell und einfach lösen, indem wir einen kürzesten Pfad in dem leicht modifizierten geometrisch dualen Graphen von G bezüglich der gegebenen Einbettung berechnen. Wir haben nun SPQR-Bäume benutzt, um einen Algorithmus zu konstruieren, der das Problem löst, auch wenn die Einbettung des Graphen nicht festgelegt ist. Dieser Algorithmus hat lineare Laufzeit bezüglich der Größe des Eingabegraphen. Wir wenden den Algorithmus für feste Einbettungen auf die dreizusammenhängenden Komponenten an, die uns der SPQR-Baum liefert. Dann kombinieren wir die erhaltenen Ergebnisse, indem wir die Struktur des SPQR-Baums ausnutzen. Das Ergebnis ist eine optimale Lösung für das ursprüngliche Problem.

Eine weitere Möglichkeit, die Eigenschaften von SPQR-Bäumen auszunutzen, um Probleme für planare Graphen zu lösen, ist die folgende: Nehmen wir an, es existiert ein lineares Programm L_1, dessen Optimallösung der Lösung unseres Problems entspricht, falls die Einbettung des Graphen festgelegt ist. Wir haben eine Methode entwickelt, mit der wir ein ganzzahliges lineares Programm L_2 konstruieren können, dessen Lösungen die Einbettungen eines planaren zweizusammenhängenden Graphen sind. Wenn wir nun L_1 und L_2 kombinieren, so erhalten wir ein gemischt-ganzzahliges lineares Programm, dessen Optimallösung der Lösung des ursprünglichen Problems für Graphen ohne festgelegte Einbettung entspricht.

Diesen Ansatz haben wir benutzt, um das Knick-Minimierungsproblem zu lösen. Eingabe dieses Problems ist ein planarer zweizusammenhängender Graph und die gesuchte Lösung ist eine orthogonale Zeichnung des Graphen mit der geringstmöglichen Anzahl von Knicken. In einer orthogonalen Zeichnung werden die Kanten durch Folgen von vertikalen und horizontalen Geradensegmenten dargestellt. Das Zusammentreffen eines horizontalen und vertikalen Segmentes der gleichen Kante wird als Knick bezeichnet.

Algorithmen, die dieses Problem lösen, wenn die Einbettung festgelegt ist, sind schon längere Zeit bekannt. Es wurde auch schon gezeigt, dass das Problem NP-schwer ist, wenn die Einbettung nicht Teil der Eingabe ist. Der Algorithmus zur Lösung des Problems bei fester Einbettung berechnet einen Fluß mit minimalen Kosten in einem speziellen Netzwerk, das von dem Graphen und der Einbettung bestimmt wird. Dieses Netzwerk-Fluß Problem kann man recht einfach als lineares Programm formulieren.

Wir haben dieses lineare Programm mit unserem neuen ganzzahligen linearen Programm kombiniert, das alle Einbettungen eines planaren zweizusammenhängenden Graphen beschreibt. Das Ergebnis ist ein gemischt-ganzzahliges lineares Programm, bei dem eine optimale Lösung einer Zeichnung des Problemgraphen entspricht, die die kleinstmögliche Anzahl von Knicken aufweist. Dieses Programm benötigt nicht die Vorgabe einer Einbettung. Wir benutzen einen kommerziellen Löser für gemischt-ganzzahlige Programme, um eine Optimallösung des so gewonnenen linearen Programmes zu berechnen. Unsere experimentellen Resultate zeigen, daß dieser Ansatz für große Graphen einem schon bekannten Branch & Bound Ansatz zur Lösung des gleichen Problems überlegen ist.

A.1 Zeichnen von Graphen

Das Ziel bei der Erstellung einer Zeichnung eines Graphen ist es, ein Bild zu erzeugen, indem man möglichst einfach ablesen kann, welche Knoten durch Kanten verbunden sind. Diese Arbeit beschäftigt sich mit zwei Problemen auf dem Gebiet des automatischen Zeichnens von Graphen.

Es gibt viele wissenschaftliche Gebiete, in denen Zeichnungen von Graphen benutzt werden, um Beziehungen zwischen Objekten zu visualisieren. Der Grund ist, daß viele Wissenschaften sich mit komplexen Systemen interagierender Objekte befassen. Um solche Systeme zu verstehen, ist es nötig, die Beziehungen zwischen den Objekten zu verstehen.

Auf dem Gebiet der Sozialwissenschaften beschäftigen sich die Wissenschaftler oft mit Netzwerken von Menschen in einer Gruppe, z.B. den Mitarbeitern in einer Firma. Die entsprechenden Beziehungen werden normalerweise in einer Zeichnung visualisiert, in der die Personen durch unterschiedliche geometrische Formen repräsentiert werden. Die Beziehungen zwischen den Personen werden durch Linien dargestellt, die die Formen verbinden. Zusätzliche Informationen sind durch Farb- und Formgebung der Linien und Formen kodiert.

Auch in den Wirtschaftwissenschaften gibt es Anwendungen für das Zeichnen von Graphen. So werden z.B. Geschäftsprozesse oft als Graphen dargestellt, wobei die elementaren Aufgaben die Knoten sind und eine Kante von einer Aufgabe zu einer zweiten aussagt, daß die erste Aufgabe vor der zweiten ausgeführt werden muß.

Eine weitere Anwendung in den Wirtschaftwissenschaften ist die Darstellung von Organigrammen. In diesen Diagrammen werden die Beziehungen von organisatorischen Einheiten beschrieben. Eine ähnliche Anwendung ist die Visualisierung von Firmenbeteiligungen (siehe Abbildung A.1).

Auch in der Informatik findet das Zeichnen von Graphen Anwendung. Besonders beim Entwurf von großen Software-Projekten und bei der Untersuchung bestehender Systeme. Ein Beispiel ist hier der Aufrufgraph von Funktionen. Die Knoten sind hierbei die Funktionen und

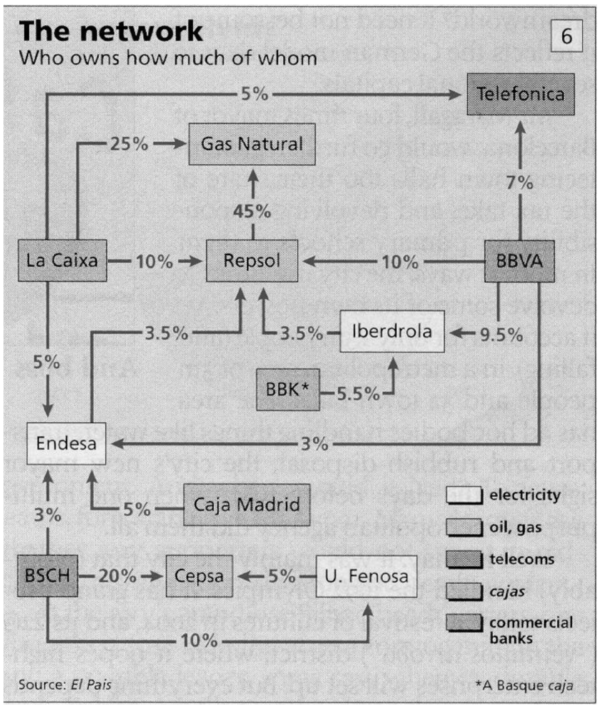


Abbildung A.1: Ein Graph, der die Firmenbeteiligungen zwischen verschiedenen spanischen Firmen zeigt.

eine Kante von einer Funktion zu einer zweiten bedeutet, dass die erste Funktion die zweite aufruft. Bei Klassendiagrammen entsprechen die Knoten den Klassen und die Kanten entsprechen Vererbungsbeziehungen oder zeigen an, dass eine Klasse die Methoden einer anderen Klasse benutzt.

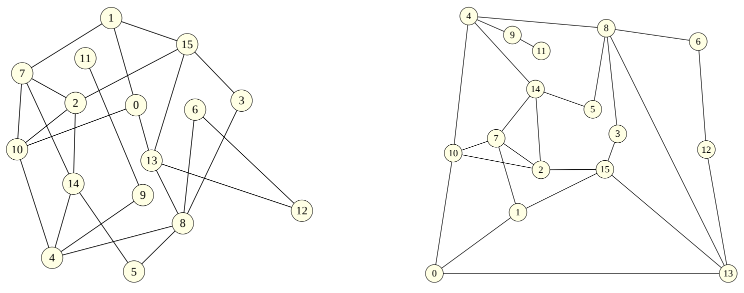
Beim Entwurf von Datenbanken wird eine graphische Entwurfsprache benutzt, die "Entity Relationship Model" genannt wird. Es handelt sich dabei auch um Graphen, bei denen die Knoten Objekte der Datenbank sind und die Kanten die Beziehungen zwischen den Objekten visualisieren.

Man kann einen Graphen auf viele verschiedene Arten zeichnen. Je nach Anwendung, muß eine Zeichnung aber meist gewisse Anforderungen erfüllen. Diese anwendungsspezifischen Anforderungen bezeichnet man als Zeichen-Konventionen. In manchen Anwendungen müssen z.B. alle Knoten als gleich große Rechtecke gezeichnet werden und die Kanten als Geradensegmente.

Aber nicht jede Zeichnung, die die Zeichen-Konvention für eine bestimmte Anwendung erfüllt, ist eine gute Zeichnung. Um leicht lesbar zu sein, sollte eine Zeichnung auch bestimmte ästhetische Anforderungen erfüllen. Die folgenden Anforderungen sind für die meisten Anwendungen am wichtigsten:

- **Geringe Anzahl von Kreuzungen:** Enthält die Zeichnung eines Graphen viele Kreuzungen, so ist es für den Betrachter meist schwierig zu erkennen, welche Knoten durch eine Kante miteinander verbunden sind. Bei den meisten Anwendungen ist eine geringe

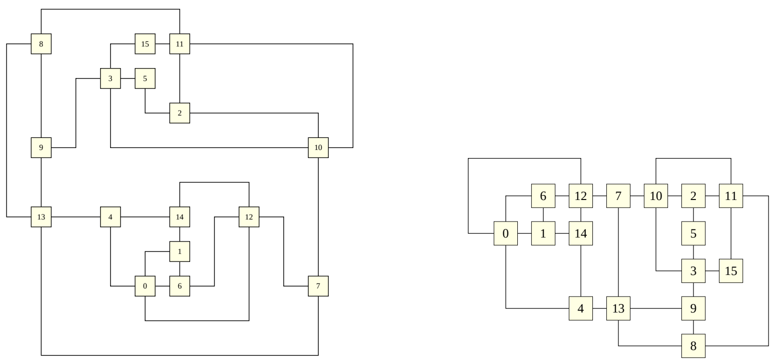
Anzahl von Überkreuzungen eine der wichtigsten Anforderungen. Abbildung A.2 zeigt zwei Zeichnungen des selben Graphen, wobei die Zeichnung auf der linken Seite wegen der vielen Kreuzungen sehr viel unübersichtlicher wirkt als die Zeichnung rechts mit nur einer Kreuzung.



(a) (b)

Abbildung A.2: Zwei verschiedene Zeichnungen des selben Graphen mit unterschiedlicher Anzahl von Kreuzungen.

- **Geringe Anzahl von Knicken:** Falls die Kanten als Folge von horizontalen und vertikalen Geradensegmenten gezeichnet werden (wie in Abbildung A.1), ist es wünschenswert, möglichst wenige Segmente pro Kante zu verwenden. Einer Kante, die aus vielen Segmenten besteht und somit viele Knicke hat, kann das Auge nicht so gut folgen wie einer Kante mit wenigen Knicken. Abbildung A.3 zeigt zwei Zeichnungen des selben Graphen, einmal mit vielen Knicken und einmal mit wenigen Knicken.



(a) (b)

Abbildung A.3: Zwei Zeichnungen des gleichen Graphen mit unterschiedlich vielen Knicken.

- **Kleine Zeichenfläche:** Wenn wir eine Zeichnung auf einem Bildschirm darstellen oder mit einem Drucker ausgeben, so haben wir nur eine begrenztes Gitter zur Darstellung der Zeichnung zur Verfügung. Dies liegt daran, dass diese Ausgabegeräte nur eine begrenzte Auflösung besitzen. Deshalb ist es wünschenswert, dass das kleinste Rechteck, das die Zeichnung eines Graphen enthält, wenige Gitterpunkte bedeckt. Wenn wir dann die Zeichnung skalieren, um sie auf dem Ausgabegerät möglichst groß wiederzugeben, so werden die Knoten und Kanten größer dargestellt, als wenn das einschließende Rechteck eine große Fläche einnimmt.

  Diesen Effekt kann man auch anhand von Abbildung A.3 erkennen. Es handelt sich bei beiden Zeichnungen um Gitterzeichnungen, d.h. sowohl die Knoten als auch die Knicke liegen auf Punkten eines Gitters. Obwohl die Zeichnung auf der linken Seite mehr Fläche der Seite bedeckt, sind die Knoten bei der Zeichnung rechts größer dargestellt. Dies liegt daran, dass für die Zeichnung links ein größeres Gitter nötig ist. Würde man beide Zeichnungen auf das gleiche Gitter zeichnen, so wäre das einschließende Rechteck für die Zeichnung links deutlich größer als für die Zeichnung rechts. Da wir die Knoten und Kanten in der Zeichnung so groß wie möglich darstellen wollen, bevorzugen wir Zeichnungen, die wenig Gitterfläche benötigen.

- **Kleine Kantenlänge:** Wenn Kanten kurz sind, folgt daraus, dass Objekte, die in Beziehung zueinander stehen, geringen Abstand haben. Dies erleichtert das Verständnis der Zeichnung. Es ist wünschenswert, dass sowohl die durchschnittliche Kantenlänge klein ist als auch die Länge der längsten Kante.

- **Symmetrie:** Manche Graphen kann man so zeichnen, dass die Zeichnung Symmetrien aufweist. Dies kann zu einem tieferen Verständnis der Beziehungen der dargestellten Objekte führen, weil eine geometrische Symmetrie immer einem Automorphismus in dem Graphen entspricht. Auch ist es wünschenswert, einen Graphen fast symmetrisch zu zeichnen, falls eine kleine Änderung des Graphen eine symmetrische Zeichnung des Graphen erlauben würde.

Es gibt noch viele andere Ästhetische Kriterien, aber die hier aufgeführten gehören zu den wichtigsten für die meisten Anwendungen. Das Problem ist, dass all diese Kriterien schwer zu optimieren sind und sich teilweise gegenseitig widersprechen. Zum Beispiel hat eine Zeichnung, die die Symmetrien maximiert oft mehr Kreuzungen, als eine Zeichnung, die die Anzahl der Kreuzungen minimiert. Abbildung A.4 zeigt ein triviales Beispiel für eine solche Situation.

In dieser Arbeit beschäftigen wir uns mit den ersten beiden ästhetischen Kriterien: der Anzahl der Kreuzungen und der Anzahl von Knicken in einer orthogonalen Zeichnung. In Kapitel 3 untersuchen wir das Problem, wie man eine neue Kanten in einen planaren Graphen einfügen kann, so dass möglichst wenige Kreuzungen entstehen. Der resultierende Algorithmus kann in einer Heuristik verwendet werden, die nicht-planare Graphen mit wenigen Kreuzungen zeichnet. In Kapitel 4 entwickeln wir einen Algorithmus, der für einen zweizusammenhängenden planaren Graphen eine orthogonale Zeichnung mit der kleinsten möglichen Anzahl von Knicken berechnet.

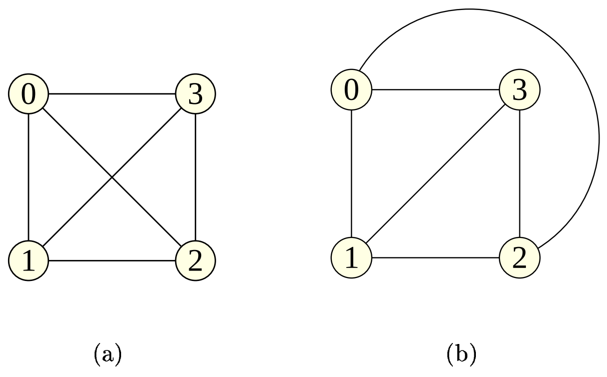


Abbildung A.4: Zwei Zeichnungen des selben Graphen. Die Zeichnung links maximiert die Anzahl de Symmetrien, während die Zeichnung rechts die Anzahl der Kreuzungen minimiert.

A.2 SPQR-Bäume

SPQR-Bäume wurden von Di Battista und Tamassia eingeführt und repräsentieren die Zerlegung eines zweizusammenhängenden Graphen in dreizusammenhängende Komponenten. Man kann sie benutzen, um alle Einbettungen eines Graphen aufzuzählen. Einbettungen beschreiben die Topologie einer planaren Zeichnung eines Graphen. Man kann eine Einbettung definieren, indem man für jeden Knoten die Reihenfolge der inzidenten Kanten im Uhrzeigersinn angibt. Die Eigenschaft von SPQR-Bäume, alle Einbettungen eines Graphen aufzählen zu können, hat uns motiviert, sie bei der Lösung der beiden Probleme, die hier betrachtet werden, zu verwenden.

Nehmen wir an, es existiert ein effizienter Algorithmus A um ein bestimmtes Problem für einen zweizusammenhängenden Graphen zu lösen, falls die Einbettung des Graphen fest steht. Dann können wir dieses Problem auch für den Fall lösen, dass die Einbettung nicht vorgegeben ist, indem wir mittels des SPQR-Baums alle Einbettungen aufzählen und für jede Einbettung den Algorithmus A aufrufen. Das Problem mit diesem Ansatz ist, dass ein zweizusammenhängender Graph exponentiell viele verschiedene Einbettungen bezüglich seiner Größe haben kann.

Wenn wir für jede Komponente des SPQR-Baums eine Einbettung festlegen, so legen wir damit auch eine Einbettung für den ursprünglichen Graphen fest. Wenn wir nur die Einbettungen für einen Teil der Komponenten festlegen, so können wir damit eine Teilmenge von Einbettungen des ursprünglichen Graphen definieren. Nehmen wir nun an, wir wollen eine Funktion über alle Einbettungen des Graphen optimieren. Können wir für eine Menge der Einbettungen, die durch teilweise Festlegung der Einbettungen der Komponenten des SPQR-Baumes definiert ist, Schranken für den Wert der zu optimierenden Funktion festlegen, so können wir damit einen Branch & Bound Algorithmus zur Optimierung der Funktion über alle Einbettungen konstruieren. Dieser Ansatz wird von (9) verwendet um die Anzahl der Knicke in einer orthogonalen Zeichnung zu minimieren.

A.3 Anwendung von SPQR-Bäumen in dieser Arbeit

SPQR-Bäume zerlegen einen zweizusammenhängenden Graphen in seine dreizusammenhängenden Komponenten. Nehmen wir an, es gibt einen effizienten Algorithmus, um eine bestimmte Funktion über den Zeichnungen eines Graphen zu optimieren, falls der Graph dreizusammenhängend ist. Wie wir schon am Beginn dieser Zusammenfassung erwähnt haben, ist eine Möglichkeit, die Funktion über die Zeichnungen eines zweizusammenhängenden Graphen zu optimieren, den bekannten Algorithmus auf die dreizusammenhängenden Komponenten im SPQR-Baum anzuwenden und die erhaltenen Lösungen zu einer Lösung für den ursprünglichen Graphen zu kombinieren.

Dies ist der Ansatz den wir verwenden, um eine neue Kante mit möglichst wenigen Kreuzungen in einen planaren Graphen G einzufügen. Falls G dreizusammenhängend ist oder die Einbettung fest steht, können wir das Problem mittels eines bekannten Algorithmus in linearer Zeit lösen. Dieser Algorithmus berechnet einen kürzesten Pfad im leicht modifizierten dualen Graphen bezüglich der gegebenen Einbettung. Dieser Pfad definiert die Kreuzungen, die zum Einfügen der neuen Kante nötig sind.

Bevor wir uns mit dem Problem beschäftigt haben, war kein polynomieller Algorithmus bekannt, der das Problem optimal lösen kann, wenn der Graph nicht dreizusammenhängend ist. Unser Resultat besagt dass es ausreicht, den bekannten Algorithmus auf bestimmte modifizierte Teilgraphen des ursprünglichen Graphen anzuwenden, die durch den SPQR-Baum definiert sind. Dadurch können wir das Problem für beliebige planare Graphen in linearer Zeit lösen.

Für das Knick-Minimierungsproblem ist die Situation ähnlich. Ist die Einbettung des Graphen gegeben, so kann man eine knick-minimale Zeichnung in polynomieller Zeit berechnen, indem man einen Fluss mit minimalen Kosten in einem speziellen Netzwerk berechnet. Außerdem ist bekannt, dass das Problem NP-schwer ist, falls über alle Einbettungen optimiert werden soll (Garg und Tamassia, 1995).

Wie schon zuvor erwähnt, existiert ein Branch & Bound Algorithmus, um die Anzahl der Knicke in der Zeichnung eines planaren zweizusammenhängenden Graphen über alle Einbettungen zu optimieren (9). Unsere Motivation war, Methoden der ganzzahligen linearen Optimierung auf dieses Problem anwenden.

Der erste Schritt war, ein ganzzahliges lineares Programm zu entwickeln, dessen Lösungen die Einbettungen des planaren zweizusammenhängenden Graphen sind. Wir haben dies erreicht, indem wir erst entsprechende ganzzahlige lineare Programme für die durch den SPQR-Baum definierten Komponenten entwickelt haben. Dann haben wir die Struktur des SPQR-Baums ausgenutzt, um aus diesen Bausteinen rekursiv ein ganzzahliges lineares Programm für den Originalgraphen zu konstruieren. Dies ist die erste Beschreibung eines ganzzahligen linearen Programmes, welche die Einbettungen eines planaren zweizusammenhängenden Graphen beschreibt.

Im zweiten Schritt haben wir das erhaltene ganzzahlige lineare Programm mit einem linearen Programm kombiniert, das für eine feste Einbettung die Menge aller orthogonalen Repräsentationen des Graphen beschreibt. Das Resultat ist ein gemischt-ganzzahliges lineares Programm, bei dem eine Optimallösung eine orthogonale Repräsentation des Graphen mit der kleinstmöglichen Anzahl von Knicken über alle Einbettungen beschreibt. Dieses Programm lösen wir mit einem kommerziellen Löser für gemischt-ganzzahlige lineare Programme.

A.4 Das Einfügen einer Kante in einen planaren Graphen

Eines der wichtigsten ästhetischen Kriterien auf dem Gebiet des Graphen Zeichnens ist die Anzahl der Kreuzungen. Eine Zeichnung mit vielen Kreuzungen ist meist nicht gut lesbar. Unglücklicherweise ist es NP-vollständig, für einen Graphen zu entscheiden, ob er mit weniger als k Kreuzungen gezeichnet werden kann (Garey und Johnson, 1983). Selbst für relativ kleine Graphen (z.B. für Graphen mit 20 Knoten), ist kein Algorithmus mit vertretbarer Laufzeit bekannt, um die minimale Anzahl von Kreuzungen zu bestimmen, mit der man den Graphen zeichnen kann.

Aus diesem Grund benutzt man Heuristiken um Zeichnungen von nicht-planaren Graphen mit wenigen Kreuzungen zu berechnen. Eine der besten Methoden, die man hierfür verwendet, ist die *Planarisierungs Methode*. Diese arbeitet in zwei Schritten: Zuerst wird eine möglichst kleine Anzahl von Kanten gelöscht, um den Graphen planar zu machen (zeichenbar ohne Kreuzungen). Dann werden die gelöschten Kanten einzeln in eine Einbettung des Graphen eingefügt und entstehende Kreuzungen durch künstliche Knoten ersetzt.

Beim Wiedereinfügen der Kanten wird jeweils eine der Kanten in den planaren Graphen eingefügt, so dass sich die Kanten des planaren Graphen nicht schneiden und die neue Kante möglichst wenige Kreuzungen erzeugt. Nach dem Einfügen der Kante, werden die Kreuzungen durch künstliche Knoten ersetzt, damit man wieder einen planaren Graphen hat, in den man die nächste Kante einfügen kann. Ein Beispiel für diesen Schritt ist in Abbildung A.5 zu sehen. Nachdem alle Kanten eingefügt sind, werden in der fertigen Zeichnung die künstlichen Knoten wieder durch Kreuzungen ersetzt.

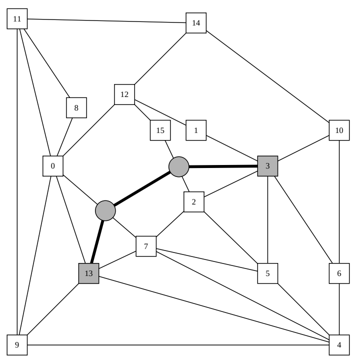


Abbildung A.5: Künstliche Knoten (hier als graue Kreise gezeichnet) wurden in den Graphen eingefügt, um die Kreuzungen zu beseitigen, die beim Einfügen der neuen Kante entstanden sind (die neue Kante verbindet die als graue Quadrate gezeichnete Knoten).

Bevor wir uns mit dem Problem beschäftigt haben, wurde das Einfügen der Kanten wie folgt durchgeführt: Man wählt eine beliebige Einbettung des planaren Graphen und berechnet einen kürzesten Pfad in dem dualen Graphen, der durch die Einbettung definiert ist. Die erhaltene Lösung garantiert, dass die Anzahl der entstandenen Kreuzungen für die gewählte Einbettung minimal ist. Allerdings kann es für andere Einbettungen bessere Lösungen geben.

Unser neuer Ansatz besteht aus zwei Algorithmen. Der erste Algorithmus wird benutzt, wenn der Problemgraph zweizusammenhängend ist. Der zweite Algorithmus kommt zur Anwendung, wenn der Graph zusammenhängend, aber nicht zweizusammenhängend ist. Der Algorithmus für zusammenhängende Graphen benutzt den Algorithmus für zweizusammenhängende Graphen. Ist der Problemgraph nicht zusammenhängend und die zu verbindenden Knoten befinden sich in unterschiedlichen Zusammenhangskomponenten, so kann man sehr einfach eine Einbettung des Graphen finden, in die man die neue Kante ohne Kreuzungen einfügen kann. Ansonsten kann man den Algorithmus für zusammenhängende Graphen auf die Zusammenhangskomponente anwenden, in der sich beide Knoten befinden.

Unsere Algorithmen berechnen genau betrachtet keine Einbettungen des Eingabegraphen sondern eine Liste von Kanten, die die einzufügende Kante kreuzen muss. Wir nennen die Liste dieser Kante einen *Einfügepfad* für die neue Kante. Um die entsprechende Einbettung zu berechnen, ist ein weiterer Schritt nötig, der aber in linearer Zeit ausgeführt werden kann: Wir fügen künstliche Knoten ein, die jede Kante im Einfügepfad in zwei Kanten aufteilen und verbinden die Knoten durch neue Kanten in der durch den Einfügepfad gegebene Reihenfolge. Dann verbinden wir den künstlichen Knoten auf der ersten Kante des Einfügepfades mit dem ersten Knoten der einzufügenden Kante und den künstlichen Knoten auf der letzten Kante des Einfügepfades mit dem zweiten Knoten der einzufügenden Kante. Das Resultat ist ein Pfad aus neuen Kanten, der beim ersten Knoten der einzufügenden Kante beginnt und beim zweiten Knoten der Kante endet (dies ist auch in Abbildung A.5 erkennbar). Dann rufen wir einen Algorithmus auf, der in linearer Zeit für einen planaren Graphen eine Einbettung berechnet. Wenn wir aus der so erhaltenen Einbettung die künstlichen Knoten und Kanten löschen, erhalten wir eine Einbettung des ursprünglichen Graphen.

Der Algorithmus für zusammenhängende Graphen konstruiert erst den *Block Baum* des Eingabegraphen bei dem die Menge der Knoten aus der Knotenmenge des ursprünglichen Graphen und den zweizusammenhängenden Komponenten (Blöcken) des Graphen besteht. Wir benutzen diese Datenstruktur, um diejenigen Blöcke zu identifizieren, deren Einbettung einen Einfluss auf die Anzahl der entstehenden Kreuzungen beim Einfügen der neuen Kante haben kann. In jeder dieser Komponenten identifizieren wir ein Paar von Knoten, welches die beiden Knoten der einzufügenden Kante repräsentiert. Dann rufen wir den Algorithmus für zweizusammenhängende Graphen für jede dieser Komponenten auf, als wollten wir die beiden Repräsentanten verbinden. Im letzten Schritt müssen wir nur noch die so erhaltenen Listen von Kanten aneinander hängen, um einen Einfügepfad für die neue Kante zu erhalten.

Der Algorithmus für zweizusammenhängende Graphen arbeitet ähnlich wie der Algorithmus für zusammenhängende Graphen. Zuerst berechnen wir den SPQR-Baum des Graphen. Jeder Knoten in diesem Baum ist mit einem speziellen Graphen assoziiert, der *Skelett* des Knoten genannt wird. Die Einbettung des Skelettes bestimmt einen Teil der Einbettung des Originalgraphen. Durch Inspektion des SPQR-Baums können wir diejenigen Knoten des Baumes bestimmen, bei denen die Einbettung des Skelettes einen Einfluss auf die Anzahl der entstehenden Kreuzungen beim Einfügen der neuen Kante hat. Wir sagen, die entsprechenden Knoten liegen auf dem *kritischen Pfad* für das Einfügen der Kante.

Es stellt sich heraus, dass nur einer der vier Knotentypen im SPQR-Baum Einfluss auf die Anzahl der Kreuzungen beim Einfügen einer neuen Kante hat. Deshalb brauchen wir uns nur mit den Knoten dieses Typs auf dem kritischen Pfad zu beschäftigen. Für jeden dieser Knoten berechnen wir nun eine Liste von Kanten, die beim Einfügen der neuen Kante gekreuzt werden müssen. Dies geschieht folgendermaßen: Erst ersetzen wir bestimmte Kanten des Skelettes des betrachteten Knoten durch Teilgraphen des Originalgraphen. Dann berechnen wir eine beliebige Einbettung des resultierenden Graphen. Da nun die Einbettung dieses Graphen feststeht, können wir den dualen Graphen berechnen. Indem wir einen kürzesten Pfad in dem dualen Graphen berechnen, erhalten wir die gesuchte Liste von Kanten. Haben wir diese Schritte für jeden der Knoten auf dem kritischen Pfad durchgeführt, so hängen wir einfach die erhaltenen Kantenlisten hintereinander, um einen Einfügepfad für die Kante zu erhalten, die in den Originalgraph eingefügt werden soll. Diese Einfügepfad gibt eine minimale Liste von Kanten an, die man beim Einfügen der neuen Kante kreuzen muss.

Ein wichtiges Element im Beweis der Korrektheit dieses Ansatzes ist, dass die Anzahl der Kreuzungen, die beim Einfügen einer Kante in einen planaren Graphen entstehen, nur von den Teilen des Graphen abhängen, die Unterteilungen eines dreizusammenhängenden Graphen sind. Man erhält eine Unterteilung eines Graphen, indem man Kanten durch Einfügen von neuen Knoten aufspaltet. Enthält ein Graph keine Unterteilung eines dreizusammenhängenden Graphen, so können wir jede beliebige Kante ohne Kreuzungen einfügen.

Eine weitere Eigenschaft von zweizusammenhängenden Graphen, die wir in unserem Algorithmus ausnutzen, ist die folgende: Enthält ein Teilgraph des Problemgraphen, der nur mittels zweier Knoten mit dem Rest des Graphen verbunden ist, keinen der beiden Knoten, die zu der neuen Kante gehören, so spielt die Einbettung dieses Teilgraphen keine Rolle für die Anzahl der Kreuzungen, die beim Einfügen der Kante entstehen. Die Argumentation ist die folgende: Gibt es einen optimalen Einfügepfad für die neue Kante, bei dem keine Kante des betrachteten Teilgraphen gekreuzt wird, so spielt die Einbettung des Teilgraphen offensichtlich keine Rolle für die Anzahl der Kreuzungen in einer Optimallösung. Andernfalls kreuzt die neue Kante in einer optimalen Lösung eine gewisse Anzahl von Kanten in dem Teilgraphen, um ihn zu durchqueren. Wir zeigen mittels vollständiger Induktion, dass die Anzahl der Kreuzungen, die nötig ist, um einen Teilgraphen zu durchqueren, unabhängig von dessen Einbettung ist.

Indem wir die beiden eben erwähnten Fakten benutzen, können wir zeigen, dass der von unserem Algorithmus berechnete Einfügepfad für die neue Kante, die kleinste mögliche Länge aller Einfügepfade hat. Es folgt, dass es keine bessere Möglichkeit geben kann, die neue Kante einzufügen, als alle die Kanten zu kreuzen, die von unserem Algorithmus als Ergebnis berechnet werden. Dies ist auch unter Berücksichtigung aller möglichen Einbettungen des Graphen wahr.

Die Laufzeit unsere Algorithmus ist linear in der Größe des Eingabegraphen. Auch die bisher verwendete nicht optimale Methode zum Einfügen einer Kante in einen planaren Graphen hat lineare Laufzeit. Der erste Grund, warum unser Algorithmus in linearer Zeit läuft, ist dass die Graphen, für die wir kürzeste Wege im dualen Graphen berechnen, disjunkt sind. Der zweite

Grund ist, dass die Vereinigung all dieser Graphen nur um einen Konstanten Faktor größer sein kann als der Originalgraph.

Wir wollten nun wissen, welche Vorteile es bringt, unseren neuen Algorithmus im Rahmen der Planarisierungsmethode zu benutzen, im Vergleich mit dem bisher verwendeten Verfahren. Obwohl unser Algorithmus eine einzige Kante optimal in einen planaren Graphen einfügt, ist es keinesfalls klar, dass dies beim Einfügen mehrerer Kanten hintereinander von großem Vorteil ist. Wie wir im Kapitel 5 zeigen, ist das optimale Einfügen von einzelnen Kanten keinesfalls immer ein optimaler Weg um mehrere Kanten einzufügen.

Glücklicherweise haben unsere Experimente auf einer Menge von Benchmark-Graphen gezeigt, dass die Verwendung unseres neuen Algorithmus im Rahmen der Planarisierungsmethode dem bisherigen Verfahren deutlich überlegen ist. Durchschnittlich werden durch Verwendung unseres Algorithmus 14% weniger Kreuzungen produziert als bei der Verwendung des bisherigen Verfahrens, bei dem die neue Kante jeweils in eine beliebige Einbettung des Graphen eingefügt wird. Es gibt einige wenige Fälle, in denen die alte Methode weniger Kreuzungen produziert als unsere neue Methode. Aber diese Fälle sind selten und der Unterschied in der Kreuzungszahl gering. Da die asymptotische Laufzeit unserer neuen Methode die gleiche ist wie bei der alten Methode, halten wir die Verwendung unseres Algorithmus in der Planarisierungsmethode für sinnvoll.

Unsere neue Methode kann nicht dazu verwendet werden, die minimale Kreuzungszahl eines Graphen zu bestimmen, selbst wenn er nach dem Löschen einer einzigen Kante planar wird. Graham Farr [2000], hat ein Beispiel gefunden, bei dem die Planarisierungsmethode mit unserem neuen Algorithmus eine Lösung berechnen kann, bei der die Kreuzungsanzahl um einen beliebig großen Faktor höher ist als in einer optimalen Lösung.

Die Ursache hierfür ist, dass die von unserem Algorithmus berechnete Lösung die minimale Anzahl von Kreuzungen nur unter all den Lösungen besitzt, in der der planare Graph, in den die Kante eingefügt wird, kreuzungsfrei gezeichnet wird. Wenn sich nun in einer optimalen Lösung Kanten dieses planaren Graphen schneiden, kann unser Algorithmus sie nicht finden.

A.5 Knickminimierung in orthogonalen Zeichnungen

In einer orthogonalen Zeichnung eines Graphen werden die Kanten als Folgen von horizontalen und vertikalen Geradensegmenten gezeichnet. Ein Beispiel ist die Zeichnung in Abbildung 1.2. Diese Art von Zeichnungen wird zum Beispiel beim Entwurf von Datenbanken mit Hilfe des Entity Relationschip Models bevorzugt. Der Punkt, wo ein vertikales Segment einer Kante sich mit einem horizontalen Segment der gleichen Kante schneidet, wird *Knick* genannt.

In orthogonalen Zeichnungen mit einer großen Anzahl von Knicken fällt es dem Betrachter schwer, dem Verlauf von Kanten zu folgen. Deshalb ist es wichtig, orthogonale Zeichnungen mit möglichst wenigen Knicken zu erzeugen. Abbildung A.3 auf Seite 151 zeigt zwei Zeichnungen des gleichen Graphen mit unterschiedlich vielen Knicken.

Es ist schon länger bekannt, dass man für eine fixe Einbettung eines Graphen in polynomieller Zeit eine Zeichnung mit der minimalen Anzahl von Knicken berechnen kann. Dies geschieht, indem man das Problem in ein Fluss-Netzwerk umwandelt, indem man dann einen Fluss mit minimalen Kosten bestimmt. Der erste Algorithmus dieser Art wurde von Tamassia entwickelt (47). Ist die Einbettung eines Graphen nicht festgelegt, so ist es ein NP-vollständiges Problem für ein gegebenes k zu entscheiden, ob man den Graphen mit höchstens k Knicken zeichnen kann (Garg und Tamassia, 1995).

Es gibt einen Branch & Bound Algorithmus, um einen planaren zweizusammenhängenden Graphen mit der minimalen Anzahl von Knicken zu zeichnen (9). In dieser Arbeit lösen wir das selbe Problem mit Hilfe von Methoden der ganzzahligen linearen Programmierung. Um dies zu erreichen, haben wir erst ein ganzzahliges lineares Programm (GLP) entwickelt, das wir für jeden zweizusammenhängenden planaren Graphen aufbauen können und dessen Lösungen den Einbettungen des Graphen entsprechen.

Im zweiten Schritt haben wir dieses GLP mit einem linearen Programm (LP) kombiniert, das für eine feste Einbettung alle orthogonalen Repräsentationen eines planaren Graphen darstellt. Eine orthogonale Repräsentation legt die Knicke einer orthogonalen Zeichnung fest aber nicht die Länge der einzelnen Geradensegmente in der Zeichnung. Die Kombination aus dem GLP und dem LP ist ein gemischt ganzzahliges lineares Programm, dessen Lösungen für einen planaren zweizusammenhängenden Graphen die Menge aller orthogonalen Repräsentationen darstellt. Mit einem Löser für gemischt ganzzahlige lineare Programme können wir eine knickminimale orthogonale Repräsentation für einen Graphen bestimmen.

Wieder nutzen wir die Tatsache aus, dass ein SPQR-Baum alle Einbettungen eines Graphen repräsentieren kann. Wie schon vorher erwähnt, ist jeder Knoten in dem Baum mit einem speziellen Graphen, seinem Skelett, assoziiert. Es ist nicht schwierig, für jedes dieser Skelette ein GLP zu konstruieren, welches die Einbettungen des Skelettes beschreibt. Der Grund ist die einfache Struktur dieser Skelette. Die binären Variablen dieser GLPs entsprechen gerichteten Kreisen in den Skeletten.

Wir haben einen Weg gefunden, die GLPs für die Skelette miteinander zu kombinieren, um die Einbettungen von größeren Teilgraphen des ursprünglichen Graphen zu beschreiben. Dies können wir rekursiv weiterführen bis wir ein GLP konstruiert haben, das die Einbettungen des ursprünglichen Graphen beschreibt.

Wir kombinieren GLPs für Teile des Graphen, indem wir die Ungleichungen der Programme *liften*. Mit liften meinen wir das berechnen von Koeffizienten für Variablen eines höherdimensionalen Problems für eine Ungleichung, die Bestandteil eines nieder-dimensionalen Problems ist. Ergebnis ist eine Ungleichung, die für das höher-dimensionale System gültig ist. In unserem Ansatz müssen wir beim Kombinieren von GLPS noch zusätzlich Ungleichungen einführen, die als Verbindung zwischen den Teil-GLPs dienen.

Es stellte sich heraus, dass die Menge der Einbettungen für das Skelett einer bestimmten Knotenart im SPQR-Baum sich als die Menge der Hamilton Touren in einem vollständigen ungerichteten Graphen darstellen läßt. Eine Hamilton Tour ist ein Kreis in einem Graphen, der jeden Knoten genau ein Mal besucht. Dadurch haben wir eine Verbindung zwischen dem Asymmetrischen Handlungsreisenden Problem (AHP) und unserem Einbettungsproblem hergestellt. Beim AHP ist ein gerichteter vollständiger Graph mit Kantengewichten gegeben und es gilt, eine Hamilton Tour mit möglichst kleinem Gesamtgewicht zu finden.

In unserem Ansatz verwenden wir die gleiche GLP-Beschreibung für die entsprechenden

Skelette im SPQR-Baum, die auch zur Lösung des AHP verwendet wird. Die Anzahl der Ungleichungen in dieser Formulierung ist aber exponentiell in der Größe des Skelettes. Deshalb haben wir einen Mechanismus entwickelt, der auch nach der rekursiven Konstruktion des Einbettungs-GLP effizient testen kann, ob ein bestimmter Vektor eine dieser Ungleichungen, die aus der AHP-Formulierung stammen, verletzt. Dadurch müssen wir nicht alle AHP-Ungleichungen in das GLP einfügen. Wir testen für jede gefundene Lösung, ob sie eine Ungleichung verletzt. Ist das der Fall, so fügen wir die betroffene Ungleichung in das GLP ein und suchen eine neue Lösung. Unsere Experimente haben gezeigt, dass dies sehr selten nötig ist.

Um nun unser ursprüngliches Knickminimierungsproblem zu lösen, kombinieren wir das GLP, das die Einbettungen des Graphen beschreibt mit einem linearen Programm (LP), das die orthogonalen Repräsentationen des Graphen für eine feste Einbettung beschreibt. Das LP ist abgeleitet von der Formulierung des Knickminimierungsproblems für feste Einbettungen als Netzwerk-Fluß Problem. Das entsprechende Netzwerk wurde schon von (9) verwendet.

Die Kombination des GLP mit dem LP erzeugt ein gemischt ganzzahliges lineare Programm, welches für einen planaren zweizusammenhängenden Graphen die Menge seiner orthogonalen Repräsentationen beschreibt. Eine Optimallösung des Programmes entspricht einer orthogonalen Repräsentation mit der minimalen Anzahl von Knicken. Wir lösen das Programm mit Hilfe von CPLEX, einem kommerziellen Löser für gemischt ganzzahlige lineare Programme.

Unsere Experimente zeigen, dass unser neuer Ansatz dem schon bekannten Branch & Bound Ansatz von (9) für große Graphen mit vielen Einbettungen überlegen ist.

A.6 Übersicht der Arbeit

In Kapitel 2 führen wir die Datenstruktur SPQR-Baum ein. Zuerst geben wir in Abschnitt 2.1 einen Überblick über die Einsatzgebiete des Baums und dessen Merkmale. Dann führen wir einige Konzepte ein und präsentieren formale Definitionen, die für die Definition des SPQR-Baums nötig sind (Abschnitt 2.2). Meist geht es dabei um den Zusammenhang von Graphen. Die eigentliche Definition des SPQR-Baum in Abschnitt 2.3 unterscheidet sich leicht von den aus der Literatur bekannten Definitionen. Um eine Anomalie bei der Definition der Wurzel zu vermeiden, führen wir erst eine Datenstruktur namens Proto-SPQR-Baum ein und definieren den SPQR-Baum als eine Erweiterung des Proto-SPQR-Baumes.

Nachdem wir ein Beispiel für die Konstruktion eines SPQR-Baums präsentiert haben, nennen wir in Abschnitt 2.4 einige Eigenschaften von SPQR-Bäumen, die in dieser Arbeit wichtig sind. Das Thema von Abschnitt 2.5 ist der Zusammenhang zwischen den Einbettungen des ursprünglichen Graphen und den Einbettungen der Skelette der Knoten des SPQR-Baums. Dies ist die Basis für die Algorithmen, die in dieser Arbeit vorgestellt werden.

Kapitel 3 beschäftigt sich mit dem Problem des Einfügens einer Kante in einen planaren Graphen mit der geringstmöglichen Anzahl von Kreuzungen. In Abschnitt 3.1 geben wir eine Einführung in das Problem. Wir stellen die Planarisierungsmethode vor, in der das Problem auftritt und präsentieren einige Resultate bezüglich verwandter Probleme.

In Abschnitt 3.2 führen wir einige notwendig Begriffe ein. Wir definieren den Begriff *Einfügepfad*, eine Liste von Kanten die angibt, welche Kanten beim Einfügen einer neuen Kante gekreuzt werden müssen. Außerdem führen wir den Begriff der *Durchquerungskosten* ein. Diese sind für Teilgraphen definiert, die mit dem Rest des Graphen nur mittels zweier Knoten verbunden sind. Man kann diese Kosten als die Anzahl der Kanten interpretieren, die man kreuzen muß, um von einer Seite des Teilgraphen zu der anderen zu gelangen.

Der Algorithmus zum Einfügen einer Kante in einen planaren zweizusammenhängenden Graphen mit der kleinstmöglichen Anzahl von Kreuzungen ist das Thema von Abschnitt 3.3. Dort beweisen wir auch seine Korrektheit und zeigen, dass die Laufzeit linear in der Größe des Graphen ist.

In Abschnitt 3.4 beschreiben wir den Algorithmus für zusammenhängende Graphen. Erst führen wir die Datenstruktur *Block-Baum* ein und stellen dann den Algorithmus vor. Dieser bestimmt erst, welche Blöcke im Block-Baum einen Einfluß auf die Anzahl der Kreuzungen beim Einfügen der neuen Kante haben und ruft dann den Algorithmus für zweizusammenhängende Graphen für die entsprechenden Blöcke auf. Wieder beweisen wir, dass der Algorithmus korrekt ist und lineare Laufzeit hat. In diesem Abschnitt geben wir auch die sehr einfache Erweiterung des Algorithmus auf nicht-zusammenhängende Graphen an.

Die experimentellen Resultate finden sich in Abschnitt 3.5. Wir vergleichen die Resultate der standard Planarisierungsmethode mit der von uns modifizierten Methode, die den neuen Algorithmus benutzt. Die Resultate zeigen, dass unser Algorithmus die Anzahl der Kreuzungen deutlich reduziert im Vergleich mit der standard Methode.

In Abschnitt 3.6 zeigen wir den Unterschied zwischen dem allgemeinen Kreuzungsminimierungsproblem und dem Einfügen einer Kante in einen planaren Graphen auf. Wir präsentieren ein Beispiel, bei dem der Unterschied der Kreuzungsanzahl in optimalen Lösungen für beide Probleme beliebig groß ist.

Das Thema von Kapitel 4 ist die Minimierung der Anzahl der Knicke in der orthogonalen Zeichnung eines planaren Graphen. Abschnitt 4.1 gibt eine Einführung in das Problem und einen Überblick über das Kapitel. Das ganzzahlige lineare Programm (GLP), das alle Einbettungen eines planaren zweizusammenhängenden Graphen beschreibt ist das Thema von Abschnitt 4.2. Erst erklären wir informell, wie das GLP aufgebaut wird. Dann definieren wir die Variablen des Programmes.

Als nächstes definieren wir das GLP für den Fall, dass der SPQR-Baum eines Graphen nur einen inneren Knoten hat. Wir unterscheiden hierbei drei Fälle, je nach dem Typ des einzigen inneren Knotens. Dabei gehen wir auf die speziellen Probleme ein, die auftreten, wenn der einzige innere Knoten den Typ *P-Knoten* hat. In diesem Fall kann man die Menge der Einbettungen seines Skelettes als die Menge der Hamilton-Touren in einem vollständigen gerichteten Graphen interpretieren. Deshalb benutzen wir die gleiche GLP-Formulierung, die auch zur Lösung des Handlungsreisenden-Problems verwendet wird.

Danach zeigen wir, wie man das GLP für Graphen berechnen kann, deren SPQR-Baum mehr als einen inneren Knoten besitzt. Dies geschieht indem man den Graphen in kleinere Teilgraphen zerlegt und dann die GLPs für diese Teilgraphen rekursiv berechnet. Dann werden die GLPs der Teilgraphen zu einem GLP für den gesamten Graphen kombiniert. Das Aufspalten in Teilgraphen wird durch Aufspalten des SPQR-Baums an einem Knoten ausgeführt.

Wir zeigen die Korrektheit unseres rekursiv definierten GLPs mittels struktureller Induktion über dem SPQR-Baum. Dadurch beweisen wir, dass alle Lösungen des GLP wirklich Einbettungen des ursprünglichen Graphen darstellen und dass alle Einbettungen des Graphen Lösungen des GLPs entsprechen.

Abschnitt 4.3 beschäftigt sich mit dem linearen Programm, das für eine feste Einbettung eines Graphen seine orthogonalen Repräsentationen darstellt. Nachdem wir eine Übersicht über den Abschnitt gegeben haben, definieren wir die Datenstruktur *orthogonale Repräsentation* als eine Einbettung des Graphen zusammen mit zwei Funktionen, die die Winkel zwischen adjazenten Kanten und die Knicke auf jeder Kante definieren. Auch geben wir ein Beispiel für eine orthogonale Repräsentation.

Orthogonale Zeichnungen realisieren orthogonale Repräsentationen indem sie den Geradensegmenten der Kanten Längen zuweisen. Wir stellen verschiedene Arten von orthogonalen Graphenzeichnungen vor. Sie unterscheiden sich in der Art, wie Knoten mit Grad größer als vier dargestellt werden. Wir führen auch den *podevsnef* Stil zum orthogonalen Zeichnen ein, den wir in unserem Algorithmus benutzen. Dann zeigen wir, wie eine orthogonale Repräsentation als Fluss in einem Netzwerk formuliert werden kann. Das resultierende Netzwerk-Fluss-Problem formulieren wir dann als lineares Programm.

In Abschnitt 4.4 kombinieren wir das lineare Programm, welches die orthogonalen Repräsentationen eines Graphen für eine feste Einbettung darstellt mit dem ganzzahligen linearen Programm (GLP), das für einen Graphen die Menge seiner Einbettungen darstellt. Das Resultat ist ein gemischt ganzzahliges lineares Programm, das für einen Graphen ohne feste Einbettung die Menge seiner orthogonalen Repräsentationen darstellt.

Den Algorithmus, den wir benutzen, um orthogonale Repräsentationen mit der minimalen Knickanzahl zu berechnen, stellen wir in Abschnitt 4.5 vor. Nach dem wir einen Überblick über den Algorithmus gegeben haben, präsentieren wir einige Modifikationen, die die Berechnung einer optimalen Lösung beschleunigen. Auch entwickeln wir ein Verfahren, das einen Eingabegraphen in einen kleineren Graphen umwandelt und die Lösung, die unser Algorithmus für den kleineren Graphen gefunden hat, wieder in eine Lösung für den ursprünglichen Graphen übersetzt.

In Abschnitt 4.6 stellen wir unsere experimentellen Resultate vor. Wir vergleichen zwei verschiedene Versionen unseres Algorithmus mit dem Branch & Bound Algorithmus von (9). Es zeigt sich, dass unser Verfahren für große Graphen mit vielen Einbettungen dem Branch & Bound Algorithmus überlegen ist. Im selben Abschnitt präsentieren wir auch einige Zeichnungen, die mit Hilfe unseres Algorithmus berechnet wurden.

In Kapitel 5 geben wir einen kurzen Überblick über die Resultate der Arbeit. Auch stellen wir mögliche Ansatzpunkte für zukünftige Forschung vor.

Curriculum Vitae

Personal Data

Name:	René Weiskircher
Date of birth:	23.07.1971
Nationality:	German
E-mail:	`weiskircher@ads.tuwien.ac.at`
URL:	`www.ads.tuwien.ac.at/people/Weiskircher.html`
Address:	TU Wien, Institut 186/1 Favoritenstraße 9-11, A-1040 Wien, Austria
Phone:	+43 1 58801-18636

Education

10/92–07/97	Studied Computer Science with secondary subject Economics at the Universität des Saarlandes, Im Stadtwald, 66123 Saarbrücken
07/96–07/97	Work on Diploma thesis at the Max–Planck–Institut für Informatik, Im Stadtwald, 66123 Saarbrücken. Title: “2–Schicht–Planarisierung bipartiter Graphen” (2–Layer–Planarization of Bipartite Graphs)
25.07.97	Diploma at the Universität des Saarlandes First examiner: Professor Dr. Kurt Mehlhorn, Max-Planck-Institut für Informatik) Second examiner: Professor Dr. Stefan Näher, Universität Halle Average: 1.2 (best possible average 1.0)
01.08.97	Begin of PhD studies (advisor Professor Dr. Mutzel)
08/97–10/97	Researcher at the Max–Planck–Institut für Informatik
11/97–09/99	Scholarship holder of the “Graduiertenkolleg Effizienz und Komplexität von Algorithmen und Rechenanlagen”
10/99–06/00	Researcher with teaching responsibilities at the Technische Universität Wien, Institut 186/1 (Institut für Computer Graphik und Algorithmen)
7/00–12/00	Lecturer at the Department for Computer Science and Software Engineering of the University of Newcastle, New South Wales, Australia
Since 1/00	Researcher with teaching responsibilities at the Technische Universität Wien, Institut 186/1 (Institut für Computer Graphik und Algorithmen)

Teaching Experience

10/97–02/98	In charge of the seminar “Ganzzahlige Optimierung” (Integer Optimization) at the Universität des Saarlandes
04/99–07/99	In charge of the exercises for the lecture “Optimierung” (Optimization)
3/00–6/00	In charge of the exercises for the lecture “Algorithmen und Datenstrukturen I” (Algorithms and Data Structures I) at the Technische Universität Wien
7/00–12/00	In charge of lecture “Introduction to Programming and Numerical Methods” at the University of Newcastle, New South Wales, Australia

Referee for International Conferences

- Seventh International Symposium on Graph Drawing, GD ’99, Stirin Castle (Prague), Check Republic, 15–19 September 1999

- Ninth International Symposium on Graph Drawing, GD ’01, Vienna, Austria, 23–26 September 2001

Publications

Proceedings

- “Two–Layer Planarization in Graph Drawing”, Petra Mutzel and René Weiskircher, Proceedings of the Ninth Annual International Symposium on Algorithms and Computation (ISAAC 98), *Lecture Notes in Computer Science (LNCS)*, volume 1533, Springer-Verlag, pages 69–79, 1998

- “Optimizing Over All Combinatorial Embeddings of a Planar Graph”, Petra Mutzel and René Weiskircher, Proceedings of the Seventh Conference on Integer Programming and Combinatorial Optimization (IPCO 99), *Lecture Notes in Computer Science (LNCS)*, volume 1610, Springer Verlag, pages 361–376, 1999

- “Computing Optimal Embeddings for Planar Graphs”, Petra Mutzel and René Weiskircher, Proceedings of the 6th Annual International Conference on Computing and Combinatorics (COCOON 2000), *Lecture Notes in Computer Science (LNCS)*, volume 1858, Springer Verlag, pages 95–104, 2000

- “Inserting an Edge Into a Planar Graph”, Carsten Gutwenger, Petra Mutzel and René Weiskircher, Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2001), ACM-SIAM, pages 246–255, 2001

Chapter in books

René Weiskircher, “Drawing Planar Graphs”. In M. Kaufmann and D. Wagner, Editors, *Graph Drawing: Methods and Models*, Lecture Notes in Computer Science (LNCS), volume 2025, Springer Verlag, 2001.

Research Reports

- “Optimizing Over All Combinatorial Embeddings of a Planar Graph”, Petra Mutzel and René Weiskircher, research report MPI-I-98-1-029, Max–Planck–Institut für Informatik, Saarbrücken, Germany
- “Inserting an Edge Into a Planar Graph”, Carsten Gutwenger, Petra Mutzel and René Weiskircher, Technical Report TR-186-1-00-01, Technical University Vienna.

Literaturverzeichnis

[AGD] AGD. *AGD User Manual (Version 1.1)*, 1999. Universität Wien, Max-Planck-Institut Saarbrücken, Universität Trier, Universität zu Köln. See also `http://www.mpi-sb.mpg.de/AGD/`.

[1] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph Drawing.* Prentice Hall, 1999.

[2] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7: 303–326, 1997.

[3] G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th Annual IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 436–441, 1989.

[4] G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In M. S. Paterson, editor, *Proc. of the 17th International Colloqium on Automata, Languages and Programming (ICALP)*, volume 443 of *LNCS*, pages 598–611. Springer-Verlag, 1990.

[5] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996.

[6] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996. ISSN 0097-5397. URL `http://epubs.siam.org/sam-bin/dbq/toc/SICOMP/25/5`.

[7] P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. *LNCS*, 1272:331–344, 1998. ISSN 0302-9743.

[8] P. Bertolazzi, G. Di Battista, and W. Didimo. Quasi upward planarity. In S. Whitesides, editor, *Graph Drawing (Proc. GD'98)*, volume 1547 of *LNCS*, pages 15–29. Springer-Verlag, 1998.

[9] P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. *IEEE Transactions on Computers*, 49(8):826–840, 2000.

[10] D. Bienstock and C. L. Monma. On the complexity of covering vertices by faces in a planar graph. *SIAM Journal on Computing*, 17(1):53–76, 1988. ISSN 0097-5397 (print), 1095-7111 (electronic).

[11] D. Bienstock and C. L. Monma. Optimal enclosing regions in planar graphs. *Networks*, 19(1):79–94, 1989.

[12] D. Bienstock and C. L. Monma. On the complexity of embedding planar graphs to minimize certain distance measures. *Algorithmica*, 5(1):93–109, 1990.

[13] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. *Journal of Comp. and Sys. Sciences*, 13: 335–379, 1976.

[14] G. Carpaneto, M. Dell'Amico, and P. Toth. Exact solution of large scale asymmetric travelling salesman problems. *ACM Transactions on Mathematical Software*, 21(4):394–409, 1995. ISSN 0098-3500. URL `http://www.acm.org/pubs/citations/journals/toms/1995-21-4/p394-carpanet%o/`.

[15] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.

[16] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *J. of Computer and System Sciences*, 30(1):54–76, 1985.

[17] T. Christof and A. Loebel. *PORTA: a POlyhedron Representation Transformation Algorithm*. Ruprecht-Karls-Universität Heidelberg, 1997.

[18] E. Dahlhaus. Linear time algorithm to recognize clustered planar graphs and its parallelization (extended abstract). In C. L. Lucchesi, editor, *LATIN '98, 3rd Latin American symposium on theoretical informatics, Campinas, Brazil, April 20–24, 1998.*, volume 1380 of *LNCS*, pages 239–248, 1998.

[19] R. Diestel. *Graph theory*. Springer-Verlag, New York, 1997. ISBN 0-387-98210-8. Translated from the 1996 German original.

[20] G. Farr. Personal Communications, 2000.

[21] S. Fialko and P. Mutzel. A new approximation algorithm for the planar augmentation problem. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 260–269, San Francisco, California, 1998.

[22] U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *LNCS*, pages 254–266. Springer-Verlag, 1996.

[23] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal Alg. Disc. Methods*, 4:312–316, 1983.

[24] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. In R. Tamassia and I. G. Tollis, editors, *Proceedings Graph Drawing '94*, volume 894 of *LNCS*, pages 286–297. Springer-Verlag, 1994.

[25] A. Garg and R. Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In S. North, editor, *Graph Drawing (Proc. GD '96)*, LNCS, erscheint. Springer-Verlag, 1997.

[26] Z. Gu, G. L. Nemhauser, and M. P. Savelsbergh. Lifted flow cover inequalities for mixed 0-1 integer programs. *Math. Program.*, 85A(3):439–467, 1999.

[27] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In J. Marks, editor, *Graph Drawing (Proc. 2000)*, volume 1984 of *LNCS*, pages 77–90. Springer-Verlag, 2001.

[28] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. In *Proceedings of the Twelth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 246–255. ACM SIAM, 2001.

[29] D. Harel and M. Sardas. Randomized graph drawing with heavy duty preprocessing. *Advanced Visual Interfaces*, pages 19–33, 1994.

[30] S. Hong. Drawing graphs symmetrically in three dimensions. In *Graph Drawing 20001*, LNCS. Springer Verlag, to appear.

[31] J. Hopcroft and R.E. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[32] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973. ISSN 0097-5397 (print), 1095-7111 (electronic).

[33] S. Hugh-Jones. Survey: Spain. *The Economist*, 2000.

[34] ILOG S.A. *ILOG CPLEX 6.5 User's Manual*, 1999. `http://www.ilog.com/products/cplex/`.

[35] M. Jünger, S. Leipert, and P. Mutzel. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design*, 17(7):609–612, 1998.

[36] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models.* Number 2025 in LNCS. Springer-Verlag, Berlin, Germany, 2001. ISBN 3-540-42062-2.

[37] G. W. Klau. *A Combinatorial Approach to Orthogonal Placement Problems.* PhD thesis, Universität des Saarlandes, 2001.

[38] G. W. Klau, K. Klein, and P. Mutzel. An experimental comparison of orthogonal compaction algorithms. In *Proceedings of Graph Drawing '00*, LNCS. Springer Verlag, 2000.

[39] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs, International Symposium, Rome*, pages 215–232, 1967.

[40] S. MacLane. A combinatorial condition for planar graphs. *Fundamenta Mathematicae*, 28:22–32, 1937.

[41] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[42] P. Mutzel and R. Weiskircher. Optimizing over all combinatorial embeddings of a planar graph. In G. Cornuéjols, R. Burkard, and G. Wöginger, editors, *Proceedings IPCO '99*, volume 1610 of *LNCS*, pages 361–376. Springer Verlag, 1999.

[43] P. Mutzel and R. Weiskircher. Computing optimal embeddings for planar graphs. In *Proceedings COCOON '00*, volume 1858 of *LNCS*, pages 95–104. Springer Verlag, 2000.

[44] G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd. *Optimization.* Number 1 in Handbooks in Operations Research and Management Science. North-Holland, New York, 1989. ISBN 0444872841 (U.S.).

[45] M. W. Padberg, T. .J. Van Roy, and L. A. Wolsey. Valid linear inequalities for fixed charge problems. *Operations Research*, (33):842–861, 1985.

[46] M. Patrignani. On the complexity of orthogonal compaction. In F. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, editors, *Proc. 6th International Workshop on Algorithms and Data Structures (WADS '99)*, volume 1663 of *LNCS*, pages 56–61. Sringer Verlag, 1999.

[47] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987. ISSN 0097-5397.

[48] R. Tamassia. On-line planar graph embedding. *Journal of Algorithms*, 21(2):201–239, 1996.

[49] J. Westbrook and R. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

[50] G. J. Woeginger. personal communications, 1998.